VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky

Diplomová práce

2012 Bc. Tomáš Lukačko

VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Grafický editor ontologií
Graphical Ontology Editor

2012 Bc. Tomáš Lukačko

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Tomáš Lukačko**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Grafický editor ontologií
Graphical Ontology Editor

Zásady pro vypracování:

Cílem této diplomové práce je vytvořit framework pro možnost sestavení vlastního grafického editoru ontologií. Zejména pak např. ontologie softwarových procesů, modelování byznys procesů apod.
Zápis ontologií v současných nástrojích je komplikovaný, neumožňuje využít výhody tohoto způsobu popisu. Cílem práce je vytvořit framework, který podpoří jednoduché vytváření vlastních grafických editorů, které zjednoduší vytváření ontologií.
Práce je určena pro více studentů, kteří budou spolupracovat na tvorbě celého frameworku.
Framework bude obsahovat zejména tyto části:
1. Vytvoření jádra prototypové architektury pro daný framework.
2. Vytvoření možnosti vygenerování si vlastního grafického nástroje např. určením způsobu zobrazování základních elementů a typů vazeb a následném vygenerování nástroje.

Na příkladu ontologií pro softwarové procesy , prozatím zachycení jen statického aspektu procesů, ověřit funkčnost frameworku.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Svatopluk Štolfa, Ph.D.**

Datum zadání: 18.11.2011
Datum odevzdání: 04.05.2012

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA Fakulta elektrotechniky a informatiky OSTRAVA 1

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlášení studenta

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

Dne: 15. 8. 2012 Podpis: .....

Poděkování

Rád bych poděkoval vedoucímu své diplomové práce panu Ing. Svatoplukovi Štolfovi, Ph. D., za odbornou pomoc a konzultaci při vytváření této práce.

Abstrakt

Tato diplomová práce se zabývá využitím ontologií pro vytváření grafických modelů sloužících k vizualizaci ontologických objektů. Práce popisuje teorii ontologií z pohledu ontologického inženýrství a obšírněji se zabývá ontologickým jazykem OWL. Za účelem tvorby grafických modelů byl vytvořen vlastní framework, který zpracovává zvolenou ontologii, načítá z ní ontologické objekty a dále je interpretuje v implementované aplikaci. Zároveň s frameworkem byl navržen nový přístup k interpretaci informací obsažených v ontologii. Tento přístup je realizován právě vytvářením modelů složených z ontologických objektů. Vytvořeným modelům jsou přiřazovány uživatelem definované grafické prvky. Tyto grafické prvky pak tvoří nový náhled na vizualizovanou ontologii.

Klíčová slova

Ontologie, OWL, OWL API, Java, Framework, model, grafické komponenty, grafický editor

Abstract

This thesis is focused on the use of ontologies for creating graphical models for the visualization of ontological objects. Thesis describes the theory from the perspective of ontology and ontological engineering and more extensively deals with the ontological language OWL. For the purpose of graphical models was created own framework, which processes the selected ontology, reads ontological objects from it and interprets them in the implemented application. Along with the framework was proposed new approach to the interpretation of the information contained in the ontology. This approach is being implemented by creating models of complex ontological objects. The models are assigned user-defined graphics. After something we can say that these grahic elements can form a new perspective on the visualized ontology.

Key words

Ontology, OWL, OWL API, Java, Framework, model, graphics components, graphics editor

Seznam použitých symbolů a zkratek

API	Application Programming Interface
GUI	Graphical User Interface
HTML	HyperText Markup Language
IRI	Internationalized Resource Identifier
KB	Knowledge Base
OSS	Open-Source Software
OWL	The Web Ontology Language
RDF	Resource Description Framework
SWT	Standard Widget Toolkit
UML	Unified Modeling Language
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium
XML	Extensible Markup Language

Seznam obrázků

Obsah

1 Úvod

Ontologické inženýrství je jedna z nových oblastí informatiky, která se zabývá ukládáním, interpretací a výměnou informací mezi stroji a lidmi. Tento obor se rozvíjí již od 80. let minulého století. Tato diplomová práce má za cíl analyzovat současné možnosti ontologií a následně navrhnout a vytvořit softwarový nástroj pro vlastní grafickou interpretaci a editaci ontologií. Tento nástroj by měl být vytvořen jako framework, který bude moct být použit pro sestavení vlastního grafického editoru ontologií.

Práce na tomto grafickém editoru byla určena pro více diplomantů. Mou prací na tomto editoru ontologií bylo především navrhnout zadaný framework a podílet se na jeho implementaci. Dále pak navrhnout a implementovat modelování vlastních grafických modelů vytvořených na základě objektů ontologie, které budou dále využity k vlastnímu zobrazení objektů ontologií a editaci ontologií. V rámci implementace jsem se také podílel na navržení a tvorbě grafického rozhraní vyvíjeného editoru.

1.1 Motivace

Ontologie jsou velice silným nástrojem využívaným ve znalostním a především ontologickém inženýrství. Není tedy divu, že pro návrh a zpracování ontologií existuje celá řada nástrojů. V současné době je asi nejrozšířenějším a nejpopulárnějším nástrojem k tomu určeným program Protégé. Ten je vyvíjen v institutu Stanford Medical Informatics. Protégé je velice silným nástrojem, díky kterému jde navrhnout, vytvářet a editovat ontologie v ontologickém jazyce OWL. Nicméně nástroj pro grafické zobrazení ontologie používá pouze jednoduchou vizualizaci. Stejně tak Protégé neumožňuje návrh ontologie pomocí jakýchkoli grafických primitiv, jak jsme tomu zvyklí například v jazyce UML. To nás přivedlo k myšlence vytvořit vlastní nástroj pro vizualizaci a následnou editaci ontologií. Takovýto nástroj by pak měl umožňovat načíst si libovolnou ontologii a následně s touto ontologií pracovat na úrovni grafických objektů. Uživateli by mělo být umožněno navrhovat tyto grafické objekty a tím vytvářet různé pohledy na ontologii. Žádnou z těchto možností práce s ontologií Protégé neumožňuje. Proto byla vytvořena toto práce, jejímž cílem je vytvořit vlastní grafický editor ontologií se všemi výše popsanými vlastnostmi.

1.2 Struktura práce

Jak již bylo zmíněno, práce je určena pro více studentů, proto se v této diplomové práci zaměřím především na nutné teoretické minimum z oblasti ontologií a následně pak pouze na svou vlastní část práce v rámci tohoto projektu. Pořadí jednotlivých a kapitol a podkapitol je zvoleno záměrně tak, aby čtenář byl vždy nejprve seznámen s problematikou a následně až s návrhem a řešením práce.

Následující dvě kapitoly jsou zaměřeny na teorii ontologií. Druhá kapitola především popisuje význam a účel ontologií, seznámí čtenáře s prvky ontologie a představí nejznámější ontologické jazyky,

pomocí kterých lze ontologie definovat a vytvářet. Třetí kapitola pak detailněji popisuje strukturu a verze ontologického jazyka OWL, který byl využit v implementaci grafického editoru.

Následující části práce jsou již zaměřeny na vlastní teorie spojené s implementací grafického editoru a samozřejmě i popis vlastní práce. Čtvrtá kapitola je zaměřena na návrh a vývoj vlastního frameworku, který je vytvořen v rámci vyvíjeného editoru ontologií. V páté kapitole je představena teorie vytváření vlastních modelů, z nichž jsou sestaveny grafické komponenty pro vizualizaci ontologie. Šestá kapitola se zaměřuje na popis uživatelského rozhraní editoru, vytváření modelů a jejich zobrazení. V poslední, sedmé kapitole, je na příkladech uvedená aplikace modelů a porovnání zobrazení ontologií.

2 Ontologie

Abychom se mohli dále bavit o využití ontologií ve vytvářeném grafickém editoru, je třeba se s nimi alespoň okrajově seznámit. Především tedy s ontologiemi znalostními využívanými v ontologickém inženýrství. Navíc je potřeba si vymezit a seznámit se s částmi a prvky ontologie, kterými se budeme dále zabývat i za účelem vytváření modelů v grafickém editoru.

2.1 Historický vývoj

Ontologie jako pojem je starý již několik set let, kdy byl význam tomuto pojmu dán ve starověkém Řecku jako jednomu z oborů filozofie. Až do druhé poloviny 19. století byly ontologie spojeny s metafyzikou a představovaly disciplínu zabývající se jsoucnem a bytím jako takovým. Více viz. (1). Nicméně tento pojem dostal nového využití právě v druhé polovině 20. století a to přesněji v 80. letech. V tomto období se začal pojem ontologie spojovat s logikou a získáváním sémantického významu ontologií. Ontologie se začaly využívat k získávání a reprezentaci znalostí. To především v oboru tzv. znalostního inženýrství, které se později rozšířilo o obor ontologického inženýrství, jenž se specializuje vyloženě na problematiku ontologií. Přesněji na aktivity spojené s procesem vývoje a metodami konstrukce ontologií (2). V celé práci již budeme hovořit pouze o informatickém pojetí ontologií.

2.2 Definice a význam ontologií

Ontologie popisuje to, co existuje a může tedy být reprezentováno v informačním, respektive znalostním systému (3). Ontologie tedy uchovává informace o tom, co existuje a dává těmto informacím význam v souvislostech nejen v rámci dané ontologie, ale i v rámci dalších ontologií a jiných informací. V literatuře se často uvádí definice ontologie jako „explicitní specifikace konceptualizace“ (4). I z této definice vyplývá, že ontologie musí být definována explicitně a dalším zjevným požadavkem na tvorbu ontologie je použití jazyka s definovanou syntaxí a popřípadě i sémantikou.

Obecně má největší význam ontologie jako souhrn informací, potažmo znalostí, které mají dopomoci lidem, strojům, nebo systémům ve vzájemném dorozumění se. V tomto ohledu se tedy neliší metafyzické pojetí od toho v informačních oborech. Dalším významem v informačních oborech je využití ontologií k pochopení čím dál tím složitějších systému. V neposlední řadě je pak významným přínosem ontologií jejich využívání ke znovupoužití doménových znalostí, nebo sjednocení a propojení odlišných terminologií (3).

2.3 Účel ontologií

Základním přínosem ontologií je tedy vytváření souhrnu informací. To za účelem vzájemného porozumění a umožnění komunikace jak mezi lidmi navzájem, tak mezi počítačovými systémy. To je jeden z nejvýznamnějších účelů pro využití ontologií. Ontologie se vytvářejí i za účelem usnadnění

návrhu znalostně orientovaných aplikací. Jako příklad těchto využití lze uvést sémantický web, který stojí na principu ontologie a snaží se, aby informace v něm obsažené měly jasně definovaný význam a rozuměl jim jak člověk, tak počítač. Aplikací ontologií za těmito účely je celá řada. Dalšími příklady mohou být vyhledávání informací ve znalostně orientovaných ontologiích, popřípadě zpracování přirozeného jazyka.

2.4 Členění ontologií

Ontologie se zabývají širokým spektrem odborných oblastí. I díky tomu lze ontologie členit z několika hledisek. V následujícím textu se pokusíme vyčlenit ontologie podle těch nejběžnějších (3).

Členění ontologií podle odborných oblastí:

1. *Terminologické (lexikální) ontologie*: tyto ontologie si lze představit jako pokročilé tezaury. Jejich využití je tedy zejména v knihovnictví a dalších oborech zaměřených na zpracování textu. Základní vlastností je využívání termínů, které nejsou dále definovány. Nejznámější terminologickou ontologií je WordNet.
2. *Informační ontologie*: Zastávají roli konceptuální vrstvy při pojmovém dotazování nad primárním zdrojem, jako například relační databází.
3. *Znalostní ontologie* jsou využívány především v oblasti umělé inteligence, kde jsou ontologie chápany jako logické teorie.

Členění ontologií podle předmětu formalizace:

1. *Doménové ontologie*: jsou nejběžněji využívaným typem ontologií. Zaměřují se pouze na konkrétní oblast, která definuje danou ontologii. Tato oblast je pak nazývána doménou. Takovouto oblastí může být například problematika podnikové činnosti a její struktury.
2. *Generické ontologie*: jsou obdobou doménových ontologií, které se ale zaměřují na obecnější koncepty reality.
3. *Úlohové ontologie* se zaměřují na procesy odvozování, nikoli na zachycení znalostí o světě. Zaujímají roli modelů řešení problémů, např. pro diagnostiku, plánování, atd.
4. *Aplikační ontologie*: je nejspecifičtějším typem ontologií, který je nejčastěji převzatý a přizpůsobený pro konkrétní aplikaci. Zahrnuje jak doménovou, tak i úlohovou část ontologie.

Členění ontologií podle míry formalizace:

Toto členění je založeno na míře použité formalizace ontologie, jak je již z názvu patrné. Patří sem ontologie striktně formální, semi-formální, či zcela neformální, které také nalézají své uplatnění. Jako neformální ontologie si lze představit například glosář, ve kterém jsou jednotlivé pojmy vysvětleny přirozeným jazykem.

2.5 Struktura ontologií

Ontologie se principielně skládají z několika základních prvků, které na sebe navazují a tvoří tak celou strukturu informací. Pro všechny typy ontologií jsou tyto prvky totožné. Rozdíl může panovat pouze

v názvosloví různých typů ontologií. Z těchto prvků jsou těmi nejpodstatnějšími tyto tři: Třída (class, koncept, kategorie, rámec), Jedinec (individual, individuum, instance), a Vlastnost (relace, funkce, slot, role, atribut). V následujícím textu se budeme postupně zabývat těmito prvky podrobněji (3).

2.5.1 Třída

Jedná se o základní prvek ontologie, který reprezentuje množinu prvků se stejnými vlastnostmi. Například v rámci pomyslné ontologie Pracovních zařazení by objekt *projekt manažer* mohl tvořit třídu sdružující všechny projekt manažery. Společnou vlastností pro všechny tyto prvky byla právě pozice projektového manažera.

Na rozdíl od tříd v objektově orientovaných modelech a jazycích nezahrnují ontologické třídy žádné metody. Interpretace třídy je spíše odvozena ve smyslu unární relace na dané doméně objektů. Třídy můžeme dále dělit na primitivní a definované. Primitivními třídami jsou ty, které nemají definované žádné podmínky nutnosti a postačitelnosti. Definované třídy jsou pak analogicky ty, které podmínky nutnosti a postačitelnosti definované mají. U tříd navíc často definujeme jejich hierarchii. Můžeme se tak v definici třídy odkazovat na nadřazené třídy, nebo podřízené třídy. Z tohoto faktu plyne i jistý předpoklad dědičnosti mezi třídami. Takováto hierarchie nabádá ke stromové interpretaci tříd, nicméně třídy mohou podporovat i vícenásobnou dědičnost a tak není stromová interpretace hierarchie tříd zcela jednoznačná.

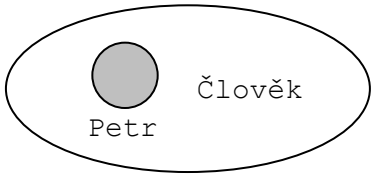
Na obrázku 1 je znázorněná třída *Člověk* reprezentující množinu lidí. V rámci této práce bude použito grafické znázornění tříd pomocí symbolu na obrázku 1.

Obrázek 1: Grafické znázornění třídy

2.5.2 Jedinec

Jedinec je z pohledu této práce klíčovým prvkem ontologií, jelikož reprezentuje právě jeden konkrétní prvek reálného světa. V rámci struktury ontologií se tento pojem chápe jako konkrétní instance dané třídy. Například v již zmíněné pomyslné ontologii Pracovní zařazení by za jedince mohl být prohlášen objekt *Petr Novák*, který není z hlediska dané ontologie považován za třídu a jeho nadřazenou třídou je třída *Projekt manažer*. Nicméně toto vysvětlení není zcela jednoznačné. Jedinec totiž může být za určitých okolností přiřazen do ontologie i bez návaznosti na konkrétní třídu. Navíc může být jedinec obsažen v několika třídách zároveň. Jinými slovy řečeno, jedinec může být definován jako instance dané třídy, ale pouze v jasném kontextu. Proto nemůžeme jednoznačně říct, že daný objekt je jedincem, či třídou. Vždy záleží na konkrétním případě a daném úhlu pohledu. V této práci budeme na jedince pohlížet především jako na konkrétní prvek z dané třídy, tedy jako na instanci třídy.

Na obrázku 2 je znázorněn objekt *Petr*, který patří do množiny dané třídou *Člověk* a je tedy jedincem z třídy *Člověk*. V rámci této práce bude použito grafické znázornění jedinců pomocí symbolu na obrázku 2.



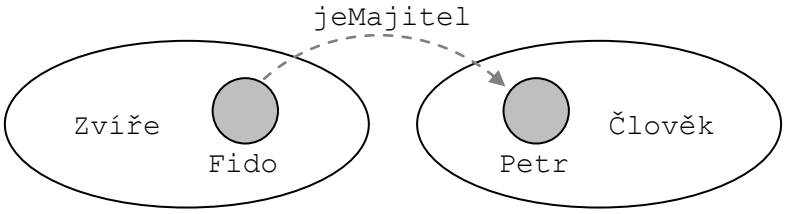
Obrázek 2: Grafické znázornění jedince

2.5.3 Vlastnost

Aby bylo v ontologii možné zachytit vztahy z reálného světa, musely být definovány vlastnosti mezi jednotlivými jedinci. Tyto vlastnosti v ontologii představují definované vztahy jak mezi jedinci, tak v podstatě i mezi třídami ontologie. Na rozdíl od objektově orientovaných modelů a jazyků stojí v ontologiích vlastnost sama o sobě bez nutné návaznosti na třídu či jedince. Tedy v ontologii je vlastnost považována za samostatný objekt i přesto, že její existence je úzce spjata s objekty jinými. Ale přesto, že vlastnost není součástí žádné třídy, lze určit definiční obor a obor hodnot vlastnosti a to jako její omezení. Samotná vlastnost je pak definována pomocí logických podmínek a návazností. Vlastnosti se navíc mohou stejně jako třídy tvořit hierarchicky a využívat dědičnosti. To znamená, že jedna vlastnost může být potomkem jiné a dohromady pak vlastnosti tvoří hierarchickou strukturu.

Společně s definicí vlastností v ontologii je třeba zmínit i jejich typy. Obecně se rozlišují dva typy. Tím běžnějším jsou objektové vlastnosti, které v ontologiích reprezentují právě výše zmíněné vztahy mezi různými objekty ontologie. Jako definiční obor se tedy využívá objektů. Odtud i název objektové vlastnosti. Druhým typem jsou datové vlastnosti. Pro ty je charakteristická definice oboru hodnot základními datovými typy. Pro představu jimi mohou být například celočíselný datový typ, reálný datový typ, výčet prvků, atd.

Na následujícím obrázku 3 je znázorněna vlastnost *jeMajitel*, definovaná nad doménou Člověk. Tato vlastnost pak spojuje jedince z třídy *Člověk* a třídy *Zvíře*. Můžeme tedy říct, že jedinec *Petr* je majitelem zvířete *Fido*.



Obrázek 3: Grafické znázornění vlastnosti

2.5.4 Axiomy

Do ontologií lze zařadit logické nebo výrokové formule. Tyto formule se nazývají axiomy, popřípadě pravidla. Na rozdíl od některých daných výrazů, které jsou přímo spojeny s příslušností k třídám nebo vlastnostem mohou axiomy mít zcela samostatné postavení nebo mohou rozšiřovat definici tříd a vlastností. Konkrétní zařazení axiomů v rámci ontologie je dáno interpretací daného ontologického jazyka. Axiomy mohou například vyjadřovat ekvivalenci tříd, nebo relací, disjunkci tříd, popřípadě rozklad třídy na podtřídy.

2.6 Ontologické jazyky

Jak již bylo výše zmíněno, pro tvorbu ontologií je potřeba využívat jazyk s jasně definovanou syntaxí a sémantikou. Proto v rámci obecného seznámení s ontologiemi v oblasti informatiky je nutné zmínit i velké množství vytvořených ontologických jazyků. Většina ontologických jazyků vznikla v poměrně krátkém období (5). V následujícím textu se pouze okrajově zmíníme o některých z nich, ale především o modernějších webových ontologických jazycích. V následující kapitole se ovšem budeme podrobněji věnovat jazyku OWL, který je stěžejním pro tuto práci.

2.6.1 Cyc

Jedná se o jeden z nejstarších projektů zaměřených na ontologie. Pro projekt byl vytvořen ontologický jazyk CycL založený na funkcionálním jazyku LISP. Vývoj projektu byl započat v roce 1984 a je rozvíjen společností Cycorp. I když projekt nebyl autory považován za ontologii, svou orientací a zaměřením odpovídá definici ontologie. Cílem bylo shromáždit všeobecné znalosti, které měly být doplňovány za pomoci znalostních systémů.

```
(#$isa #$BarackObama #$UnitedStatesPresident)
```

Toto tvrzení zapsáno v jazyce CycL tvrdí, že Barack Obama je prezidentem Spojených Států Amerických (6).

2.6.2 Ontolingua

Při tvorbě tohoto jazyka, který vznikl na počátku 90. let pod vedením T. Grubera, bylo cílem vytvořit jakýsi mezi-jazyk určený k výměně informací mezi znalostními systémy. Z toho vyplývají omezené možnosti odvozování v tomto jazyce. Ovšem i přesto byl jazyk Ontolingua řadu let velice populární pro tvorbu ontologií bez závislosti na konkrétním znalostním systému. Jazyk Ontolingua je obdobně jako CycL koncipován jako nadstavba funkcionálního jazyka. Tímto jazykem je KIF (Knowledge Interchange Format), který využívá syntaxi funkcionálního jazyka LISP (3).

2.6.3 SHOE a Ontobroker

V tomto případě se jedná o zástupce tzv. historických webových ontologických jazyků (5).

Jazyk SHOE (Simple HTML Ontology Extensions) vznikl v roce 1996 jak první jazyk pro účely přidání sémantiky k webovým stránkám. Jedná se o výrazně jednodušší ontologický jazyk. Jazyk

poskytuje možnost vkládání metadat o objektech přímo do struktury HTML stránek. Zároveň je do stránek vkládána i samotná ontologie, která definuje sémantiku vkládaných metadat. Ta se nachází na začátku struktury stránky.

Projekt Ontobroker je na rozdíl od jazyka SHOE důsledně centralizován. Ale základní myšlenkou je rovněž obohacování HTML stránek o sémantická metadata. Projekt Ontobroker předpokládal vytvoření centralizované architektury, kterou by představoval ontologický server. Ten měl spravovat ontologie a umožňovat jejich editaci pouze kvalifikovaným uživatelům. Sbírání dat z anotovaných stránek pro ontologii nemělo být náhodné. Pro sběr dat byli určeni poskytovatelé informací, patřící k určité komunitě. Z tohoto důvodu nepoužívá Ontobroker jednotný jazyk pro ontologie a anotace. Jazyk pro anotace je prostým obohacením HTML o dodatečné atributy obsahující metadata pro ontologii. Jazykem pro ontologie je propracovaný logický kalkul F-logic (3).

2.6.4 RDF

Na konci 90. let byla potřeba vytvořit formát, díky kterému by bylo možné reprezentovat informace o webových zdrojích. Z této motivace vznikl formát RDF (Resource Description Framework), který je standardem konsorcia W3C pro reprezentaci webových zdrojů.

Se vznikem formátu RDF vznikl i sémantický jazyk na RDF orientovaný. Tímto jazykem je RDFS (RDF Schema), který rovněž vznikl pod iniciativou W3C. Tento jazyk doplňuje formát RDF o základní ontologické prvky. Těmi jsou především třídy, vlastnosti, možnosti nastavení definičních oborů a oborů hodnot. RDFS rovněž umožňuje vytvářet třídy i relace hierarchicky (7).

2.6.5 DAML+OIL

Jazyk DAML+OIL vznikl sloučením dvou různých dříve vyvíjených ontologických jazyků. Jazyk DAML-ONT (DARPA Afen Mark-up Language) byl vytvořen jako sémantický jazyk pro formát RDF, který by měl větší vyjadřovací sílu než jazyk RDFS. Jazyk OIL (Ontology Interface Layer) vznikl na principu využití deskriptivní logiky (8).

Základním principem tohoto jazyka je reprezentace tříd buď za pomoci definovaného názvu třídy, tzv. URI, nebo za pomoci logického výrazu. Tyto logické výrazy jsou tvořeny konstrukty, které jsou v jazyce definovány. Konstrukty lze libovolně skládat a díky tomu vytvářet složitější výrazy. Obsah ontologie pak tvoří axiomy, které jsou vytvořeny nad výrazy reprezentující třídy.

3 OWL

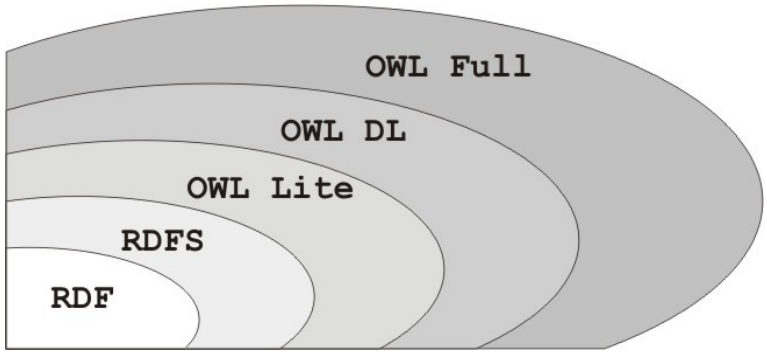
V této kapitole se dostáváme k popisu nejmladšího z řad ontologických jazyků. Pro účely této práce se jedná o stěžejní jazyk především proto, že implementace grafického editoru využívá ontologií popsaných právě v tomto jazyce. Z tohoto důvodu si jazyk OWL zaslouží větší míru pozornosti. V následujícím textu budou popsány především jeho verze, charakteristiky a v neposlední řadě i jeho syntaxe.

Ontologický jazyk OWL, jak už bylo výše zmíněno, patří k nejmladším ontologickým jazykům vůbec. Jeho vznik se datuje rokem 2002 a je stejně jako formát RDF vyvinut a podporován konsorciem W3C. Tento jazyk je primárně určen pro tvorbu ontologií využívaných v oblasti sémantického webu. Řadí se proto stejně jako DAML+OIL, nebo RDFS k moderním webovým ontologickým jazykům.

Jednou z typických charakteristik jazyka OWL je vysoká míra expresivnosti se zachováním kompatibility s formátem RDF a značkovacím jazykem XML. Jazyk je tedy bohatší na sémantiku než např. jazyk DAML+OIL (9).

3.1 Verze OWL

Zvláštností jazyka OWL je jeho postupný vývoj ve třech různých verzích. Tyto verze jsou navrženy podle potřeb uživatelů jazyka a jejich nároků na vytvářenou ontologii. Verze se od sebe liší především různou úrovní schopností a různými nároky na implementaci konkrétního projektu. Tyto verze se nazývají OWL Lite, OWL DL a OWL Full (9). Rozvrstvení verzí a v podstatě rozvrstvení architektury OWL si lze jednoduše znázornit na následujícím obrázku 4.



Obrázek 4: Znázornění vrstvené architektury OWL

3.1.1 Verze OWL Lite

Tato verze jazyka OWL je jeho nejjednodušší variantou. OWL Lite je určen především pro tvorbu méně strukturovaných ontologií s jednoduchými omezeními. Verze OWL Lite má oproti dvěma následujícím verzím mnoho omezení. Ty snižují jeho vyjadřovací sílu, ale na druhou stranu je jazyk mnohem jednodušší a efektivněji se zpracovává. Díky tomu je mnohem snazší vytvářet nástroje pro tuto verzi jazyka OWL.

3.1.2 Verze OWL DL

Tato verze jazyka OWL v sobě přináší standart deskripční logiky. Odtud je i název DL. Charakteristickými prvky této varianty jsou efektivní zpracování jazyka a dobrá výpočetní podpora. Není zde ale zajištěna plná kompatibilita s RDF a RDFS, zároveň není možné aplikovat logické výrazy jazyka na sebe navzájem. Nicméně OWL DL na rozdíl od verze Lite obsahuje všechny konstrukce jazyka OWL, které ale mohou být použity s jistým omezením.

3.1.3 Verze OWL FULL

Nyní se dostáváme k nejširší verzi jazyka OWL. Verze OWL Full v sobě zahrnuje již všechny konstrukce jazyka OWL a je možné je libovolně kombinovat s RDF a RDFS výrazy. Zajišťuje plnou kompatibilitu s formátem RDF a jazykem RDFS a je tedy platným tvrzením, že každý projekt vytvořen v RDF je také platným v jazyce OWL Full. Cenou za tyto vlastnosti je ale složitost OWL Full, která vede k nemožnosti úplné výpočetní podpory pro odvozování a k velké složitosti zpracování.

3.2 Struktura OWL dokumentu

Ontologie vytvořené pomocí jazyka OWL se skládají se základních trojic, které již předepisuje RDF. Definice trojic je tedy přenesena z RDF i do OWL a těmito trojicemi jsou stejně jako v RDF objekt, predikát a subjekt. Nejběžnější syntaxi pro zápis ontologií v jazyce OWL je OWL/XML. Následující popis struktury vychází z (9).

Každý dokument zapsaný v OWL by měl začínat definovanou hlavičkou, která obsahuje především název ontologie – IRI, informaci o verzi dokumentu, popřípadě informace o vnořených ontologiích. Jelikož se jedná o zápis v XML, dokumenty vždy uvozuje hlavička typická pro XML dokumenty. Samotná hlavička pak může mít následující podobu.

```
<Ontology xmlns="http://www.w3.org/2002/07/owl#"
      xml:base="http://www.semanticweb.org/ontologies/2012/2/Ontology133283
      8197796.owl"
      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:xml="http://www.w3.org/XML/1998/namespace"
      ontologyIRI="http://www.semanticweb.org/ontologies/2012/2/Ontology133
      2838197796.owl">
      <Prefix name="owl" IRI="http://www.w3.org/2002/07/owl#"/>
      ...
</Ontology>
```

V příkladu si můžeme všimnout, že definice hlavičky jsou deklarovány jako atributy párové značky *Ontology*. Tato značka vždy ohraničuje celou definici ontologie v jazyce OWL. Bezprostředně za hlavičkou pak následují deklarace samotné ontologie. Ty se skládají z komponent jazyka OWL.

3.2.1 Zápis třídy v OWL

V jazyce OWL existují třídy předdefinované, anebo námi vytvořené. Typickým příkladem předdefinované třídy je třída **owl:Thing**. Tato třída je systémová a jedná se o nejobecnější třídu, která je zastoupena v každé ontologii. Jak bylo uvedeno výše, třídy se v ontologiích udržují v hierarchické struktuře. Systémová třída **owl:Thing** je pro každou ontologii základní třídou, do které spadají všechny dále definované třídy. Je to tedy rodič všech dalších tříd v ontologii. Opakem třídy **owl:Thing** je další systémová třída **owl:Nothing**.

Třídy jsou v OWL vždy jednoznačně určeny svým názvem - IRI. Třída může být dále definována omezením vlastností třídy, výčtem prvků, sjednocením, průnikem, nebo doplňkem více jiných tříd.

```
<Declaration>
      <Class IRI="#Clovek"/>
</Declaration>
```

Tento jednoduchý zápis ukazuje na příkladu deklaraci třídy *Clovek*. Deklarace veškerých konstruktů v jazyce OWL se vždy uvádějí v párové značce *Declaration*.

```
<SubClassOf>
      <Class IRI="#Clovek "/>
      <Class abbreviatedIRI=":Thing"/>
</SubClassOf>
```

Hierarchický zápis tříd je v OWL uveden deklarací *SubClassOf* . Na tomto příkladu lze vidět definici třídy *Clovek*, která je podtřídou systémové třídy *owl:Thing*.

3.2.2 Zápis jedince v OWL

Jedinci rovněž patří k základním konstruktům OWL. Jelikož se v následujících kapitolách této práce budeme jedinci zabývat, je potřeba se zmínit i o jejich zápisu ve struktuře jazyka OWL i přesto, že z ontologického hlediska nejsou jedinci tím nejhlavnějším prvkem a některé jazyky dokonce jejich deklarace nedovolují.

```
<Declaration>
<NamedIndividual IRI="#Petr"/>
</Declaration>
<ClassAssertion>
      <Class IRI="#Clovek"/>
      <NamedIndividual IRI="#Petr"/>
</ClassAssertion>
```

Na příkladu jde vidět definice jedince jako *NamedIndividual*. Tento jedinec je vytvořen jako instance třídy *Clovek*, pomocí deklarace *ClassAssertion*.

V OWL nelze jakkoliv explicitně říct, že někteří jedinci jsou stejní, nebo různí. Je třeba tuto informaci do ontologie zavést pomocí deklarace *SameIndividual*, pokud se jedná o totožné jedince, nebo

DifferentIndividual, pokud se jedná o jedince zcela různé. V následujícím zápisu lze vidět, že jedinec *Petr* je totožný s jedincem *Otec1* a rozdílný s jedincem *Matka1*.

```
<SameIndividual>
      <NamedIndividual IRI="#Otec1"/>
      <NamedIndividual IRI="#Petr"/>
</SameIndividual>
<DifferentIndividuals>
      <NamedIndividual IRI="#Matka1"/>
      <NamedIndividual IRI="#Petr"/>
</DifferentIndividuals>
```

3.2.3 Zápis vlastnosti v OWL

Obecný princip vlastností v ontologiích byl již popsán výše, proto se zde zaměříme pouze na rozlišení typů vlastností v OWL a uvedeme si příklady zápisu. V OWL lze rozlišovat čtyři typy vlastností. Těmi jsou vlastnosti objektové, datatypové, anotační a ontologické.

Objektové vlastnosti jsou v OWL spojením mezi jednotlivými jedinci ontologie. Jejich zápis v OWL je zprostředkován pomocí definice *ObjectPropertyAssertion*. Pokud chceme definovat inverzní vlastnost, pak je použita definice *NegativObjectPropertyAssertion*. V následujícím příkladu je uvedena deklarace objektové vlastnosti *jeMajitel*, která ukazuje spojitost mezi jedinci *Petr* a *Fido*.

```
<Declaration>
      <ObjectProperty IRI="#jeMajitel"/>
</Declaration>
<ObjectPropertyAssertion>
      <ObjectProperty IRI="#jeMajitel"/>
      <NamedIndividual IRI="#Petr"/>
      <NamedIndividual IRI="#Fido"/>
</ObjectPropertyAssertion>
```

Datatypové vlastnosti jsou spojením jedince s hodnotou určitého datového typu. Zápis tohoto typu vlastností je za pomoci definice domény *DataPropertyDomain* a rozsahu *DataPropertyRange*. Na dalším příkladu je jednoduchý zápis vlastnosti *maVek*.

```
<Declaration>
      <DataProperty IRI="#maVek"/>
</Declaration>
<DataPropertyDomain>
      <DataProperty IRI="#maVek"/>
      <Class IRI="#Clovek"/>
</DataPropertyDomain>
<DataPropertyRange>
      <DataProperty IRI="#maVek"/>
      <Datatype abbreviatedIRI="xsd:integer"/>
</DataPropertyRange>
```

Poslední dva typy vlastností v OWL zmíníme pouze okrajově, protože jejich uplatnění není tak časté a nejsou podstatné pro tuto práci. Anotační vlastnosti mohou přidávat ke konstruktům v OWL jako jsou třídy a jedinci, popřípadě i k celé ontologii, dodatečné informace. Ontologické vlastnosti zase definují vztahy mezi různými ontologiemi.

4 Framework pro editor ontologií

Cílem této práce je navrhnout, zrealizovat a popsat řešení vlastního grafického editoru. V úvodu již bylo zmíněno, že nejrozšířenější editor ontologií Protégé, určený k editaci nad jazykem OWL, je sice velice silným nástrojem, ale jeho omezení spočívá především ve vizualizaci výstupní ontologie. Je to pochopitelné, protože tento nástroj je primárně určen k tvorbě, editaci a ukládání ontologií. Vizuální stránka ontologií je proto podružná (10). V našem případě ale nechceme ontologii využít jen jako znalostní systém. Chceme využít znalostí definovaných v ontologii k jejich vizuální prezentaci, kterou si navíc sami definujeme. Pomocí takovéto vlastní vizuální prezentace by bylo možné nahlížet na jednu ontologii z různých pohledů. Navíc by se mohl následně lišit i přístup k práci s ontologií a k její editaci.

Abychom mohli přistoupit k vlastnímu modelování vizuálních reprezentací prvků ontologie, musíme nejprve samotnou ontologii zpracovat a vytvořit znalostní bázi, která nám bude objekty definované v ontologii reprezentovat a poskytovat. Toto je jedním z prvních důvodů, proč je nutné před vytvořením grafického editoru uvažovat nad mezivrstvou mezi samotnou ontologií a její prezentací. Tuto mezivrstvu pro nás bude představovat framework, určený ke zpracování a přípravě dat v ontologii pro jejich prezentaci. Další motivací pro přiklonění se k tvorbě vlastního frameworku je možnost jej znovu použít za jiným, byť obdobným účelem.

Po prostudování teorie ontologií a ontologických jazyků jsme přistoupili k variantě využít pro náš framework výše detailně popsaný jazyk OWL. Volba padla na OWL hned z několika důvodů. Tento jazyk je momentálně jedním z nejexpresivnějších, nejrozšířenějších, nejpodporovanějších a z hlediska softwarových nástrojů i jedním z nejdostupnějších jazyků pro práci s ontologiemi. Ve chvíli, kdy máme vybrán a detailně popsán jazyk, pomocí kterého budeme chtít ontologie zpracovávat a víme, že editor, který je cílem této práce, bude založen na námi vytvořeném frameworku, je třeba přistoupit k návrhu realizace.

4.1 OWL API

Pokud máme v plánu jakýmkoli způsobem pracovat s ontologií, je třeba, aby vyvíjený software byl schopen kteroukoli ontologii navrženou v jazyce OWL zpracovat. Toho lze docílit dvěma variantami. Buď vlastním navržením API, které zpracování OWL zajistí, nebo je druhou variantou využít některého z již k tomuto účelu navržených a realizovaných API. Obsahem této práce nemá být návrh vlastního API pro zpracování ontologií v jazyce OWL, nýbrž navržení nástroje pro práci s ontologiemi. Proto jsme přistoupili k druhé variantě a z dostupných API jsme vybrali OWL API.

OWL API je OSS sada knihoven implementovaných v programovacím jazyce JAVA určených pro práci s ontologiemi vytvořenými v jazyce OWL. OWL API je určeno pro vytváření, práci a serializaci OWL ontologií. Poskytuje třídy a metody pro načítání a ukládání souborů OWL, dotazování nad ontologií a práci s datovým modelem ontologie (11). Tuto sadu knihoven jde z hlediska vývoje aplikací využívat dvěma způsoby:

- vývoj komponent a uživatelských rozhraní pro program Protégé
- vývoj samostatných aplikací v jazyce JAVA pro práci s ontologiemi

Jak je vidět ze způsobů využití, pro které je API určeno, OWL API je vhodným nástrojem pro zpracování ontologie a načtení informací o ní.

4.1.1 Reasoner

Reasoner je klíčovým prvkem pro práci s ontologií v OWL prostřednictvím OWL API. Ve skutečnosti téměř všechny dotazy z ontologie OWL jsou prováděny pomocí reasonera. Je tomu tak proto, že znalosti v ontologii nemusí být explicitně dány a při zpracování ontologie nemusí být některé vztahy a souvislosti zřejmé. Reasoner proto vyhodnocuje dotazy na ontologii vždy tak, že vyvodí patřičné důsledky mezi všemi objekty v ontologii s přihlédnutím k aktuálnímu stavu ontologie. Díky využití reasonera by měly být získané výsledky dotazů na ontologii vždy správné. OWL API obsahuje různá rozhraní pro přístup k OWL reasonerům. Pro přístup k reasoneru přes OWL API je potřeba využít poskytované rozhraní implementované v OWL API.

Reasonerů, které podporuje OWL API je celá řada (12):

- FaCT++.
- HermiT
- Pellet
- RacerPro (via OWLLink)

Pro využití reasonerů v OWL API je potřeba stáhnout příslušné knihovny a umístit je do námi vyvíjeného projektu. Všechny výše zmíněné reasonery implementují rozhraní *OWLReasoner*. Pomocí rozhraní *OWLReasonerFactory* je pak provedeno spouštění a vyhodnocování dotazů na ontologii.

Na začátku vývoje editoru bylo experimentováno s několika různými reasonery. Nakonec byl vybrán jako nejvhodnější reasoner HermiT, který je v implementaci použit.

4.1.2 Využití OWL API v implementaci

Využití nástrojů z knihovny OWL API je v naší implementaci opravdu široké. Tím nejpodstatnějším je jistě načtení a zpracování vstupní ontologie. Toho je docíleno pomocí implementace rozhraní *OWLOntologyManager*, které zajistí načtení ontologie. Ontologii lze načíst jak ze zdroje dostupného na internetu, tak z lokálního zdroje. Je třeba podotknout, že díky tomuto rozhraní je možné si vlastní ontologii i zcela vytvořit. V následující ukázce z implementace lze vidět definice tříd *OWLOntology* a *OWLReasoner*. Pomocí instance třídy *OWLOntology* je zajištěno načtení ontologie ze známého IRI ontologie. Instance třídy *OWLReasoner*, reprezentující reasonera v aplikaci, se pak postará o zpracování informací dostupných ze samotné ontologie. Toto zpracování ale reasoner provádí až v případě zavolání metody *precomputeInterfaces*.

```
OWLOntologyManage rmanager = OWLManager.createOWLOntologyManager();
OWLOntology ont = manager.loadOntologyFromOntologyDocument(DOCUMENT_IRI);
OWLReasonerFactory reasonerFactory = new Reasoner.ReasonerFactory();
```

```
OWLReasoner reasoner = reasonerFactory.createReasoner(ont, config);
reasoner.precomputeInferences();
```

Samotné načtení ontologie a její zpracování na úrovni OWL API by samozřejmě bylo pro naši práci nedostatečné. Je potřeba umět s načtenou ontologií i pracovat a získávat z ní konkrétní instance ontologických objektů. Práci nám velice zjednodušuje výše zmíněný reasoner, který jíž po zpracování ontologie na požádání vyhodnocuje vztahy a vlastnosti těchto objektů. V implementaci budeme pracovat převážně se základními konstrukty jazyka OWL, těmi jsou, jak již bylo řečeno, třídy, jedinci a vlastnosti.

Třídy jsou v OWL API reprezentovány třídou *OWLClass*. Pro jejich načtení je využit reasoner. Jelikož víme, že ontologie umožňují hierarchické rozdělení tříd, rozhodli jsme se tuto hierarchii zachovat i při jejich načítání a pro další práci s nimi. Rozhraní *OWLReasoner* nám umožňuje načíst podtřídy dané třídy. Toho jsme využili právě k hierarchickému načítání tříd. Pomocí rekurzivní metody *GetOWLNodes* se doptáváme vždy na podtřídy dané třídy. Další předpoklad pro úspěšné využití tohoto rekurzivního načítání je použít jako vstupní objekt nejvýše postavenou třídu v hierarchii tříd. Již jsme se zmínili, že právě v OWL je touto třídou *owl:Thing*. Následující ukázka kódu metody *GetOWLNodes* ukazuje jednoduchost tohoto rekurzivního přístupu.

```
private static void GetOWLNodes(Node<OWLClass> node, OWLClass parent,
DefaultMutableTreeNode jTreeParent){
      if (node.isBottomNode())
            return;
      DefaultMutableTreeNode jNode = GetOWLNode(node, parent, jTreeParent);
      OWLClass cl = node.getRepresentativeElement();
      for (Node<OWLClass> child:
      reasoner.getSubClasses(node.getRepresentativeElement(), true)){
            GetOWLNodes(child, cl, jNode);
      }
}
```

Jedince reprezentuje v OWL API třída *OWLNamedIndividual*. Opět pro jejich načítání využíváme možností reasonera. V této práci se zaměřujeme především na jedince jako instance tříd. Jedince pak načítáme pomocí dotazování na instance konkrétní třídy. Následující ukázka kódu je příklad takovéhoto dotazu na reasonera. Metoda reasonera *getInstance,* kterou využíváme, vrací sadu všech jedinců dané třídy.

```
reasoner.getInstances(owlClass, true).getFlattened();
```

Posledním z podstatných ontologických objektů, které především využíváme v naší práci, jsou objektové vlastnosti definované v ontologii. Pro načtení jednotlivých vlastností definovaných v ontologii jsme využili metody dostupné z rozhraní *OWLOntology*. Pro potřeby vizualizace objektových vlastností nebylo třeba zachovávat hierarchickou strukturu, a proto načítáme vlastnosti lineárně. Metoda, která je vidět na následujícím příkladu, *getObjectPropertySignature* vrací sadu všech objektových vlastnosti.

```
ont.getObjectPropertiesInSignature(true);
```

Další konstrukce, které využívají metod a tříd poskytnutých OWL API již nebudou uvedeny v této části práce. Některé zajímavé či podstatné aplikace využívající OWL API budou dále zmíněny v následujícím textu.

4.2 Návrh frameworku

Nyní se dostáváme k popisu samotného navrženého frameworku. Ten byl kompletně implementován v programovacím jazyce Java. Framework se skládá ze tří logických částí a je rozdělen do tří balíčků.

Rozdělení do tří logických částí vyplývá z požadavků na framework, které jsou zmíněny v úvodu této kapitoly. První část je zaměřena na samotnou ontologii, tvoří takzvanou znalostní bázi, popřípadě knowlage base. Druhá část se stará o interpretaci a poskytování dat z této knowlage base jiným částem frameworku a aplikacím na frameworku vystavěným. Poslední třetí část se zabývá modelováním komponent k vizualizaci. Tyto komponenty pak budou představovat vstup pro samotnou vizuální část výsledné aplikace. Na obrázku 5 lze vidět toto rozdělení frameworku v rámci pohledu na celou vyvíjenou aplikaci.

Obrázek 5: Grafické znázornění logických částí grafického editoru ontologií

Z programového hlediska je ale framework rozdělen do tří balíčků, přičemž tyto tři části částečně reprezentují výše zmíněné pomyslné rozdělení. Těmito třemi balíčky jsou:

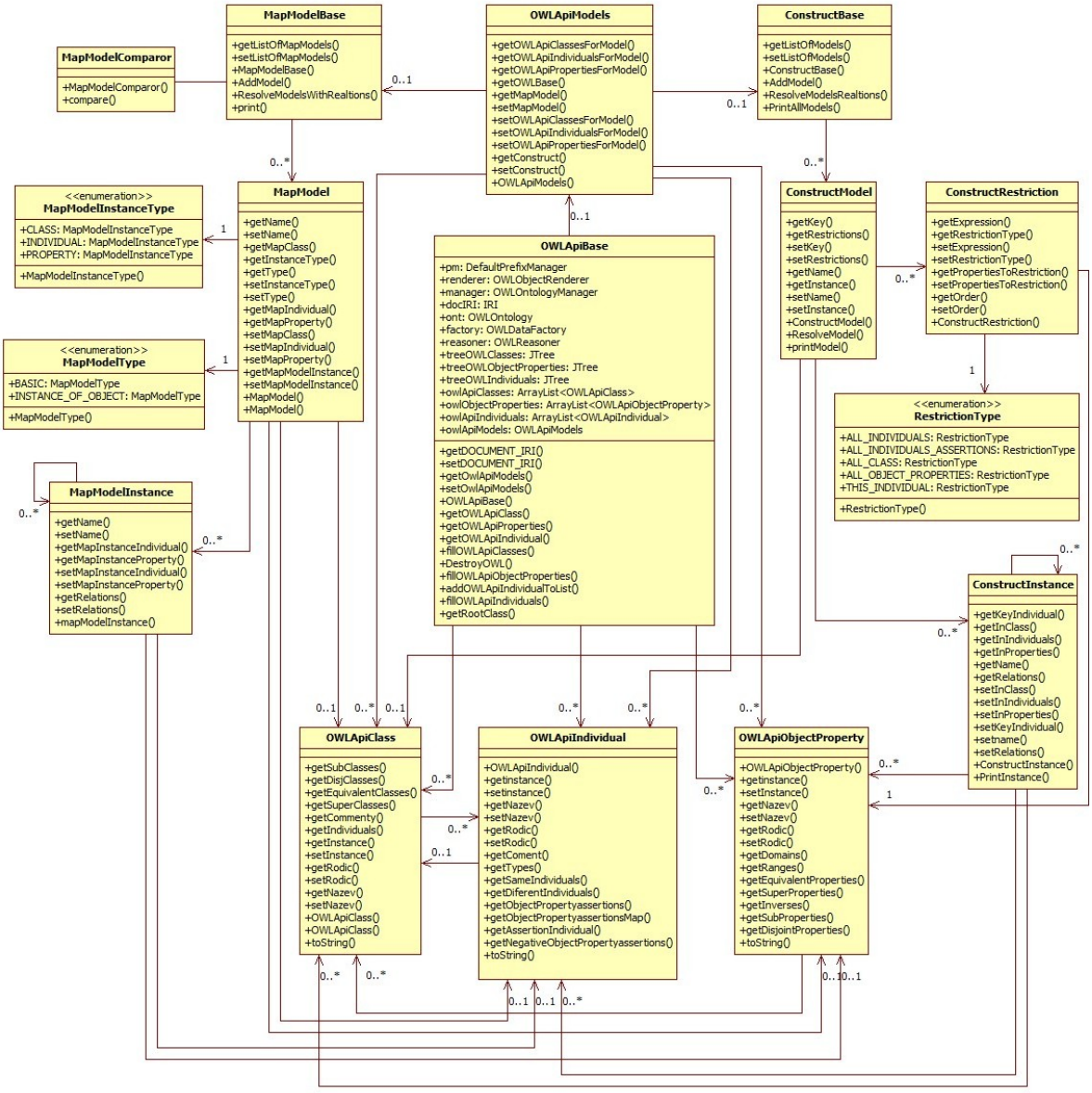
- OWLApi
- MapModel
- Construct

Balíček *OWLApi* v sobě zahrnuje knowlage base a třídy vytvořené pro zprostředkování dat mezi knwolage base a ostatními částmi frameworku. Samotnému chápání a implementaci knowlage base se budeme podrobněji věnovat v následujícím textu. Mimo to obsahuje i třídu *OWLApiModels*, která slouží k uchovávání a poskytování vytvořených komponent k vizualizaci.

Balíček *MapModel* je určen pro vytváření jednoduchých vizuálních komponent. Tyto komponenty se vytvářejí pomocí mapování jednotlivých prvků ontologie na vizuální reprezentaci. Balíček *Construct* tvoří obdobně jako předchozí balíček vizuální komponenty, nicméně tyto komponenty se neskládají

pouze ze samotných objektů, ale z kombinací několika různých objektů ontologie. Oba dva tyto balíčky spolu s principem vytváření vizuálních komponent budou popsány v následující kapitole.

Na následujícím třídním diagramu lze vidět kompletní navržené třídy a jejich spojitosti pro implementaci frameworku.



Obrázek 6: UML Třídní diagram implementovaného frameworku

4.3 Vytvoření znalostní báze

V předchozí podkapitole byla zmíněna jako jedna z částí framworku tzv. znalostní báze. V následujícím textu budeme tuto znalostní bázi nazývat jejím mezinárodním označením knowlage

base. Pojem knowlage base je v širším kontextu chápán jako speciální databáze, ve které jsou uloženy speciální informace pro určitou doménu uživatelů. Ontologie mohou představovat strojově čitelnou strukturu pro takovouto knowlage base a spolu s informacemi uloženými v ontologii jí mohou tvořit. Bavíme-li se o knowlage base v kontextu námi vyvíjeného frameworku, mluvíme tak o souhrnu všech informací uložených a poskytovaných zpracovanou ontologií.

V případě vyvíjeného frameworku je knowlage base reprezentována daty, které poskytuje využívaná OWL API. Její přímou reprezentací v aplikaci je tedy objekt instance objektu *OWLOntology*. Tento objekt je ve frameworku udržován ve třídě *OWLApiBase*, která pro framework představuje jakousi základní třídu, o kterou se opírají všechny zbylé objekty. Kterákoli ontologie, která bude načtena do frameworku bude zpracována právě touto třídou a informace v ní obsažené z ní budou dostupné.

Načtení dat z knowlage base zajišťují rovněž metody naší třídy *OWLApiBase*. Jsou jimi metody *fillOWLApiClasses*, *fillOWLApiObjectProperties* a *fillOWLApiIndividuals*. Úkolem těchto metod je načíst data ze znalostní báze a vytvořit reprezentace objektů obsažených v knowlage base, potažmo v ontologii, do připravených struktur.

Pro urychlení a zjednodušení práce s knowlage base byly navrženy třídy reprezentující konstrukty OWL. Tyto třídy mají za úkol zprostředkovávat informace o jednotlivých typech objektů dalším částem aplikace. Těmito typy objektů ontologie jsou třídy, jedinci a objektové vlastnosti. Všechny informace o konstruktech definovaných ontologií jsou dostupné z těchto tříd pomocí veřejných metod. Tyto metody ale nezpřístupňují uchovávané statické informace, nýbrž obsahují struktury dotazů na samotnou ontologii a reasonera. Tím pádem výsledky dotazů na ontologii odpovídají aktuálnímu stavu. Nevýhodou je vyšší náročnost na běh frameworku. Na obrázku 6 lze tyto třídy jednoznačně identifikovat. Jsou jimi *OWLApiClass*, *OWLApiIndividual* a *OWLApiObjectProperty*. Spolu s třídou *OWLApiBase* tvoří tyto třídy základ frameworku.

OWLApiClass

Třída *OWLApiClass* reprezentuje jednu konkrétní třídu dané ontologie. Mimo popisných informací, jako je název třídy, udržuje především instanci reprezentované ontologické třídy jako instanci objektu *OWLClass* z OWL API. Obdobně se uchovává i odkaz na rodiče této třídy. To z důvodu snadnějšího a rychlého přístupu k rodičovským prvkům v hierarchickém rozdělení ontologických tříd. Dále jsou definovány metody, které vrací informace o dané ontologické třídě. Těmito metodami jsou:

- ***getSubClasses*** – Metoda jenž vrací seznam všech podtříd dané třídy, dle hierarchického rozdělní tříd v ontologii. Pomocí dotazu na reasonera jsou navráceny všechny podtřídy v aktuálním kontextu aplikace. Seznam těchto podtříd reprezentuje generický list instancí třídy *OWLApiClass*.
- ***getDisjClasses*** – Metoda pomocí dotazu na reasonera zjistí seznam všech disjunktních tříd vůči aktuální třídě. Disjunktní třídy jsou takové, jejichž instance jedinců nemohou být zároveň instancemi jedinců třídy dotazované. Obdobně jako inverze u inverzní vlastnosti, i disjunktnost tříd není dána explicitně. Je jí třeba definovat u každého objektu představující třídu ontologie, je-li to vhodné. Metoda opět vrací seznam disjunktních tříd jako generický list instancí třídy *OWLApiClass*.

- ***getEquivalentClasses*** – Tato metoda vrací seznam tříd, u nichž je definována ekvivalence s danou třídou. Ekvivalenci tříd lze definovat jako seznam všech tříd, které mají přesně stejné instance jedinců. Návratovou hodnotou je generický list instancí *OWLApiClass* reprezentující ekvivalentní třídy dané třídy.
- ***getSuperClasses*** – Seznam super class lze chápat jako soubor všech nadřazených tříd pro danou třídu. Jak bylo v obecném popisu ontologických tříd zmíněno, hierarchie tříd v ontologiích nelze vždy chápat jako stromovou strukturu. Jedna třída může být podtřídou více tříd. Proto i v tomto případě je návratovou hodnotou metody generický list instancí třídy *OWLApiClass* reprezentující všechny nadřazené třídy náležící třídě, na níž byl vznesen dotaz.
- ***getCommenty*** – Tato metoda vrací všechny metadata, která jsou definována pro danou třídu. Jedná se ve většině případů o popis třídy. Metoda vrací seznam textových řetězců, které představují popisné komentáře definované pro danou třídu.
- ***getIndividuals*** – Nakonec se dostáváme k seznamu jedinců, kteří zde vystupují jako instance dané třídy. Tato metoda tedy vrací pouze ty jedince, kteří jsou definování ve zvolené ontologii jako instance dané třídy. Návratovou hodnotou je generický list třídy *OWLApiIndividual* odpovídající instancím jedinců dané třídy.

OWLApiIndividual

Třída *OWLApiIndividual* v našem frameworku reprezentuje konkrétního jedince obsaženého v ontologii. Tohoto jedince si lze představit jako instanci třídy a to ať už definované třídy, nebo třídy předdefinované. Třída *OWLApiIndividual* obsahuje svůj popisný název. V třídě se také uchovává odkaz na instanci objektu jedince v knowlage base jako instance třídy *OWLNamedIndividual*. Pro jednoduchost a urychlení dotazování na objekty je zde uchováván i statický odkaz na třídu *OWLApiClass*, jejíž instancí dané individuum je. Pro získání dalších informací o daném jedinci jsou definovány tyto metody:

- ***getTypes*** – Všichni jedinci v OWL jsou potomky třídy, ať už definované, nebo předdefinované. Tato metoda vrací třídu, nebo třídy, jejichž instancí je daný jedinec. Jelikož jedince může být současně instancí jedné a více tříd, je jako návratová hodnota generický list typu OWLApiClass.
- ***getSameIndividuals*** – V ontologii mohou být definováni jedinci jako totožní. Tato metoda vrací generický list těchto totožných jedinců reprezentovaných třídou *OWLApiIndividual*.
- ***getDiferentIndividuals*** – Rovněž mohou být v rámci ontologie dva jedinci označeni jako zcela odlišní. Metoda vrací seznam těchto jedinců. Návratovou hodnotou této metody je generický list třídy *OWLApiIndividual* zcela odlišných jedinců s daným jedincem.
- ***getObjectPropertyassertions*** – Jak bylo v teorii ontologií popsáno, je možné mezi jedinci definovat souvislosti vytvořením vlastností mezi dvěma jedinci. Pro vysvětlení vazby je potřeba si vztah obou jedinců vysvětlit na zápisu axiomu, který vlastnost definuje.

  $$\mathrm{ObjectPropertyAssertion}(\mathrm{OPE}, a_1, a_2) = \left((a_1)', (a_2)'\right) \in (\mathrm{OPE})^{\mathrm{OP}}$$

  kde a_1 a a_2 jsou jedinci, OPE je axiom definující danou vlastnost a OP je vlastnost objektů.
  Z tohoto zápisu lze vyčíst, že pořadí jedinců vystupujících v definici objektové vlastnosti není komutativní. Proto první jedinec vystupuje ve vlastnosti jako nositel instance vlastnosti a je jí spojen s druhým jedincem. Jako návratová hodnota této metody jsou tedy všechny objektové

vlastnosti, které byly definovány pro daného jednotlivec a tento jedinec ve vlastnosti vystupuje jako nositel instance dané vlastnosti.

- ***getNegativeObjectPropertyassertions*** – Analogicky k předchozí metodě funguje i tato metoda. V ontologii je možné definovat i zápornou vlastnost, kterou řekneme, že daná individua nejsou spojeny danou vlastností. Zápis této vazby je dán následující definicí.

$$\text{NegativeObjectPropertyAssertion}(\text{OPE}, a_1, a_2) = \left((a_1)', (a_2)'\right) \notin (\text{OPE})^{\text{OP}}$$

Výstupem této metody je tedy generický list záporných vlastností pro daného jedince.

- ***getObjectPropertyassertionsMap*** – V návaznosti na metodu *getObjectPropertyassertions* byla vytvořena tato metoda, jelikož pouhá znalost všech vlastností svázaných s daným jedincem je nedostačující. Je potřeba znát i jedince, se kterými je daný jedinec spojen těmito vlastnostmi. Metoda proto vytváří hash tabulku, ve které v roli klíče působí daná vlastnost a v roli hodnoty je generický list všech jedinců touto vlastností spojených s daným jedincem. Pro názornost následuje samotná implementace této metody.

```
public Map<OWLApiObjectProperty, ArrayList<OWLApiIndividual>>
      getObjectPropertyassertionsMap() {
      objectPropertyassertionsMap = new HashMap<
            OWLApiObjectProperty,
            ArrayList<OWLApiIndividual>>();
      for (OWLObjectProperty objProp:
            OWLApiBase.ont.getObjectPropertiesInSignature(true)) {
            for (OWLNamedIndividual ind:
            OWLApiBase.reasoner.getObjectPropertyValues(this.instance,
                  objProp).getFlattened()) {
                  OWLApiObjectProperty myApiProperties =
                        OWLApiBase.getOWLApiProperie(objProp.getIRI());
                  OWLApiIndividual myApiIndividual =
                        OWLApiBase.getOWLApiIndividual(ind.getIRI());
                  ArrayList<OWLApiIndividual> individualList =
                        objectPropertyassertionsMap.get(myApiProperties);
                  if (individualList == null) {
                        individualList = new ArrayList<OWLApiIndividual>();
                  }
                  individualList.add(myApiIndividual);

                  objectPropertyassertionsMap.put(myApiPropertie,
                        individualList);
            }
      }
      return objectPropertyassertionsMap;
}
```

Návratovou hodnotou této metody je tedy výše popsaná hash tabulka, obsahující dvojce vlastností a seznamu individuí.

- ***getAssertionIndividual*** – Tato metoda byla vytvořena pro interní potřeby frameworku. Jako vstupní parametr metody je zde konkrétní instance třídy *OWLApiObjectProperty* a návratovou hodnotou jsou instance jedinců, spojených s daným jedincem danou objektovou vlastností.

OWLApiObjectPropertie

Tato třída slouží pro reprezentaci vlastností definovaných v ontologii, nikoliv jejich instancí. Třída opět obsahuje popisný název a instanci konkrétní vlastnosti z knowlage base. Rovněž obsahuje odkaz na rodičovskou vlastnost, protože, jak už bylo výše řečeno, i vlastnosti mohou být tvořeny hierarchicky. Metody zprostředkovávající informace o dané vlastnosti jsou následující:

- ***getDomains*** – Vlastnosti mohou být omezeny jen na určité domény, například jedinci spojení touto vlastností, mohou být jen z dané třídy. Metoda vrací seznam domén jako generický list instancí *OWLApiClass*.
- ***getRanges*** – Rozsah (range) určuje, s jakým typem jedinců může být daný jedinec spojen pomocí vlastnosti, která má rozsah definován. Návratovou hodnotou této metody je generický list *OWLApiClass*, který obsahuje všechny třídy určující rozsah dané vlastnosti.
- ***getEquivalentProperties*** – V rámci ontologie lze definovat některé vlastnosti za ekvivalentní. Tato metoda vrací seznam všech ekvivalentních vlastností k dané vlastnosti jako generický list datového typu *OWLApiObjectPropertie*.
- ***getSuperProperties*** – I metody, jak již bylo několikrát zmíněno, mohou být vytvořeny hierarchicky. Metoda *getSuperProperties*, obdobně jako analogická metoda v třídě *OWLApiClass*, vrací generický list všech nadřazených vlastností pro danou vlastnost. Datový typ generického listu je opět *OWLApiObjectPropertie*.
- ***getSubProperties*** – Obdobně jako předchozí metoda, tato vrací seznam všech podřízených vlastností. Návratovou hodnotou je opět generický list datového typu *OWLApiObjectPropertie*.
- ***getInverses*** – V ontologii mohou být definovány inverzní vlastnosti. Tato metoda vrací seznam všech inverzních vlastností pro danou vlastnost.
- ***getDisjointProperties*** – Metoda vrací seznam všech disjunktních vlastností vůči dané vlastnosti jako generický list.

5 Teorie a implementace modelů

Již v úvodu čtvrté kapitoly bylo zmíněno, že cílem této práce je nejen vytvořit framework schopný načítat a interpretovat data z ontologie, ale rovněž vytvořit vizuální komponenty určené k vykreslení v grafickém editoru. Tyto vizuální komponenty budeme v kontextu frameworku souhrnně označovat za modely.

Pro potřeby vytváření modelů založených na libovolné ontologii reprezentované prostřednictvím jazyka OWL jsme navrhli a vytvořili tři typy modelovacích přístupů. Tyto tři typy by měly svým rozsahem být dostačující pro modelování jakéhokoli typu modelů nad zvolenou ontologii. Při tvorbě modelů jsme vycházeli z předložených příkladů, které nebylo možné namodelovat v ontologii a k jejich pochopení bylo potřeba zevrubnějších znalostí ontologie. Při vytváření těchto modelů je sice na tvůrce modelů kladen vyšší nárok na znalost objektů ontologie a vztahů mezi nimi, nicméně po vytvoření modelů by měl být výsledek srozumitelný i pro uživatele neznalého detailních vztahů v ontologii. Tento způsob vytváření modelů by tedy měl být vhodným nástrojem i pro práci s ontologiemi softwarových procesů, nebo pro modelování byznys procesů.

5.1 Teorie vytváření modelů

Na první pohled by se dalo říct, že pro vytvoření modelu by mohla stačit pouhá definice grafické reprezentace pro vykreslování jednotlivých konstruktů v ontologii, tak jak tomu bývá ve většině modelovacích přístupů. Nicméně tento přístup aplikován na jednotlivé instance ontologických konstruktů vede k přílišné nepřehlednosti a podstata takovéhoto grafického znázornění ontologie pak ztrácí smysl. Při hlubším pohledu na obecný model ontologií je zřejmé, že s pouhým vykreslováním podle typu konstruktu si nelze vystačit, pokud chceme zobrazit data obsažená v ontologii přehledně. Popřípadě máme-li v plánu například některé objekty od sebe odlišit, i přesto že jsou to zástupci stejného druhu konstruktů. Pak je tedy třeba si definovat, co požadujeme vykreslovat a jak si přejeme toto vykreslení provést. To nás přivádí k vytváření modelů.

Principielní základ vytváření modelů spočívá v definování a vyčlenění objektů, respektive množin objektů dané ontologii, se kterými chceme následně pracovat. Tyto objekty pak tvoří základ pro modelování a vyhodnocování modelů. Definicí objektů ontologie pro konkrétní modelování pak vzniká model mapování. Navíc je potřeba v grafickém rozhraní těmto modelům definovat vizualizaci, podle které se budou modely vykreslovat. Tomuto procesu budeme říkat mapování modelů. Výsledkem mapování je pak množina definovaných modelů a následně množina modelů grafických komponent. Objekty, které byly vybrány do modelu mapování, ale nebyl pro ně definován model, jsou po vyhodnocení z vykreslování vyloučeny.

V rámci vyhodnocování vztahů mezi modely a vykreslování grafických komponent bylo nutné vytvořit jistý typ priorit mezi různými typy modelů. Podle těchto priorit budou vytvořené modely vyhodnocovány a zadané grafické komponenty vykreslovány. Důvodem pro zavedení priorit je nutnost zabránění redundantního zobrazení zvolených prvků ontologie a jednoznačné rozpoznání vztahů mezi jednotlivými modely grafických prvků.

Pro modelování je ve frameworku vytvořena celá řada objektů. Základním objektem ovšem je třída *OWLApiModels*, která udržuje instance vytvořených modelů a vyhodnocuje vztahy mezi jednotlivými instancemi různých typů modelů. Tato třída slouží v rámci frameworku jako rozhraní mezi knowlage base a jednotlivými modely. Její instance je proto vytvářena a udržována v mateřské třídě *OWLApiBase*.

5.2 Typy modelů

Při návrhu vytváření modelů jsme si nedokázali vystačit jen s jedním typem modelovacího přístupu. Po předložení některých složitějších příkladů na výstup modelování bylo jasné, že modelovacích přístupů bude muset být použito hned několik. V naší práci jsme proto vytvořili tři takovéto přístupy a tedy tři druhy modelů.

5.2.1 Modely typu Basic

Jedná se o nejjednodušší typ modelování s nejnižší prioritou rozhodování při vytváření a vykreslování modelů. Modely typu Basic vytváří ekvivalentní zobrazení pro obecné prvky konstruktů OWL. Princip tohoto mapování tedy spočívá pouze v definici grafických komponent pro všechny konstrukty jazyka OWL, se kterými v rámci aplikace pracujeme. Těmito elementy jsou třídy, jedinci a objektové vlastnosti.

5.2.2 Modely typu MapModel

Tento typ modelování slouží pro vytváření jednoduchých reprezentací konkrétních instancí konstruktů ontologie. Pro toto modelování může být zvolena jakákoliv instance konstruktů ontologie. A to tedy jak jednotliví jedinci a vlastnosti, tak i konkrétní instance třídy.

Modely typu MapModel mají druhou nejnižší prioritu vyhodnocování. To znamená, že model tohoto mapování má vyšší prioritu než mapování obecných prvků, ale mapování může být za určitých okolností zanedbáno.

5.2.3 Modely typu Construct

Tento typ modelování je založen na vytvoření složitějších modelů, které nahrazuje nedostatky předcházejících modelů i nedostatky samotného jazyka OWL. Tento typ modelování má nejvyšší prioritu a modely daného typu jsou tedy vyhodnocovány jako první.

Vytváření modelu spočívá ve výběru domény modelu, tzv. klíče a následné rozšíření modelu o dodatečné podmínky. Model jako takový je tvořen pouze doménou modelu a podmínkami, jež jej rozšiřují. Ovšem samotný model netvoří výsledné grafické komponenty. Těmi jsou až instance daného modelu.

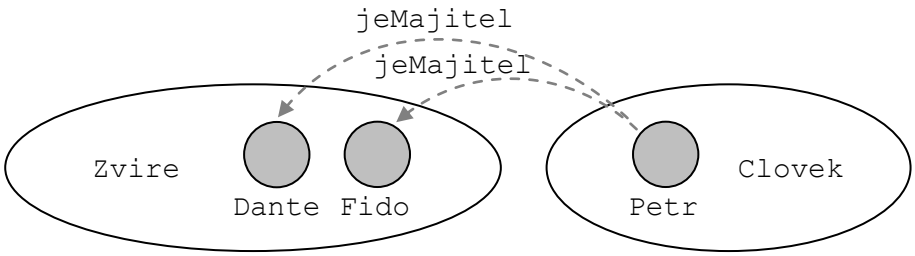
5.3 Řešení modelů Basic a MapModel

V této části kapitoly se dostáváme k širšímu popisu modelování modelů typu Basic a MapModel. Tato dvě modelování jsou záměrně sloučena do jedné části, jelikož se z principielního hlediska jedná o obdobný přístup k vytváření modelu.

5.3.1 Stavba modelu a tvorba instancí

Modely typu Basic i modely typu MapModel mají stejný principielní základ. Model je tvořen výběrem jednoho prvku a k tomuto prvku je přiřazeno konkrétní grafické zobrazení. Tím je vytvořena grafická komponenta. Vyhodnocování vztahů je rovněž totožné pro oba dva typy modelů.

Příklad 1: Na úvod si uvedeme příklad, na kterém budou demonstrovány oba dva druhy modelů. Pro modelování jsme si vybrali třídu *Zvire*, jedince *Petr*, *Fido* a Dante a vlastnost *jeMajitel*. Jedinec *Petr* je instancí třídy *Clovek*, *Fido* a *Dante* jsou instancí třídy *Zvire*. Objekty mají definovány vztahy podle následujícího diagramu:

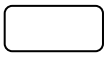


Obrázek 7: Vztahy mezi objekty pomyslné ontologie

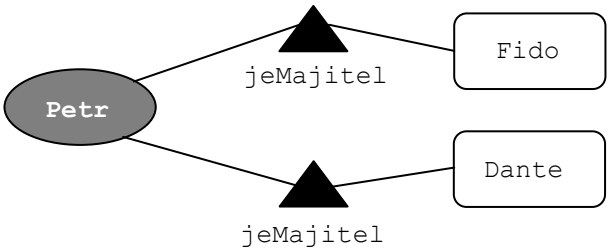
Pro vybrané prvky byly vytvořeny následující modely grafických komponent:

1. Model typu Basic: Jedinci budou zobrazeni následovně:

2. Vlastnost *jeMajitel* bude zobrazena následovně:

3. Instance třídy *Zvíře* budou zobrazeny následovně:


Výsledkem tohoto mapováni je následující zobrazení:



Obrázek 8: Výsledek vykreslení definovaných modelů z příkladu

Vysvětlení výsledného vykreslení modelu bude obsaženo v následujícím textu, kde bude odkazováno na uvedený příklad.

Modely typu Basic

Jak už bylo řečeno, modely typu Basic jsou nejjednodušším typem modelů. Tyto modely v podstatě nahrazují základní zobrazení konstruktů OWL pro vykreslování. Toto mapování má svým způsobem zahrnut i ontograf, který vykresluje prvky ontologie, obsažený v editoru ontologií Protégé. Pro potřeby modelování lze vytvořit model z následujících konstruktů OWL:

- Třída (Class)
- Jedinec (Individual)
- Vlastnost (Object Propertie)

Jelikož byla tomuto typu modelů přidělena nejnižší priorita, bude toto mapování bráno v potaz, až pro objekty, kterým nebyl vytvořen jiný model, ale byly vybrány pro vykreslení. Samotné vykreslení mapovaných objektů odpovídá definovaným grafickým komponentám.

Jak jde vidět na obrázku 6 v příkladu 1, byl jedinec *Petr*, jenž nebyl obsažen v žádném jiném definovaném modelu, zobrazen dle modelu 1, tedy podle mapování pro zobrazení jedinců. Naopak jedinci *Fido* a *Dante* jsou zobrazeni podle definovaných modelů s vyšší prioritou.

Vztahy mezi modely pro tento typ mapování jsou vykreslovány klasicky dle definice vlastností v dané ontologii. Vykreslení relace je tedy dáno vztahem mezi jednotlivými jedinci.

Modely typu MapModel

Modely typu MapModel byly navrženy za účelem vytvoření si vlastních zobrazení pro vybrané objekty ontologie. Modely tohoto typu jsou obdobné typu Basic, vytváří jednoduchou vazbu mezi konkrétním objektem ontologie a vizuálním zobrazením. Tím je vytvořen model. Pro model může být vybrána kterákoli instance konstruktů v OWL. Jak už bylo řečeno, modely typu MapModel mají druhou nejnižší prioritu vyhodnocování. Jsou tedy upřednostňovány před modely typu Basic.

Samotné vyhodnocování modelů pak probíhá rozdílně podle typu konstruktu, jehož instance byla modelována. Pro každý model je vytvořena instance modelu. Pro instance tříd, tedy jednotlivce, model

odpovídá instanci. Pro model třídy představují instance všechna individua, který byla vybrána do modelu mapování. Tyto instance jsou pak vizualizovaný podle grafické reprezentace daného modelu. Vlastnosti vytvářejí své instance až na základě vyhodnocení vztahů daného modelu. Pokud je vlastnost mezi jednotlivými modely obsažena, je vytvořena daná instance vlastnosti a její zobrazení rovněž odpovídá grafické reprezentaci daného modelu. Vztahy mezi modely pro tento typ mapování jsou tedy vykreslovány klasicky dle definice vlastností v dané ontologii.

Jak jde vidět na obrázku 6 v příkladu 1, byl pro vlastnost *jeMajitel* vytvořen model 2, jako vazba mezi touto vlastností a danou vizuální reprezentací. Instance modelu tedy byly vykresleny mezi každým jedincem, který tuto vlastnost obsahoval. V příkladu 1 byl rovněž vytvořen model pro třídu *Zvire*. Ve výběru objektů pro modelování byly i dva jedinci z této třídy. Tito jedinci pak byli přiřazeni jako instance daného modelu 3. třídy *Zvire* a vykresleni s odpovídající vizuální reprezentací.

5.3.2 Implementace modelů

Samotná implementace těchto dvou typů modelů je v našem frameworku realizována stejnými objekty. To především z důvodu podobnosti přístupů pro oba dva typy modelů. Tyto modely jsou implementovány v balíčku *OWLApi.MapModel*, který obsahuje všechny třídy potřebné k realizaci modelů.

Základní třídou je *MapModelBase*, která především udržuje generický list se seznamem všech vytvořených modelů. Jejím primárním úkolem je ale především vyhodnocení instancí modelů a vztahů mezi nimi. Toto vyhodnocování provádí metoda *ResolveModelsWithRelations*. Vyhodnocování modelů je pak provedeno na základě specifického algoritmu, který by se dal popsat v krocích takto:

1. Všechny definované modely jsou seřazeny podle předem daného pořadí dle jejich typu. Toto pořadí je následující:
    a. Modely typu MapModel definované pro konkrétního jedince
    b. Modely typu MapModel definované pro konkrétní vlastnost
    c. Modely typu MapModel definované pro konkrétní třídu
    d. Modely typu Basic definované pro jedince
    e. Modely typu Basic definované pro vlastnosti
    f. Modely typu Basic definované pro třídy

    Toto seřazení zajišťuje vlastní porovnávač implementovaný ve třídě *MapModelComparator*.
2. Jsou vytvořeny kopie vybraných prvků pro modelování.
3. Jsou vytvořeny instance modelů typu MapModel.
4. Jsou vytvořeny instance modelů typu Basic.
5. Jsou vyhodnoceny vztahy mezi instancemi na základě vlastností mezi jednotlivými jedinci obsaženými v instancích modelu.

Třída *MapModel* udržuje definované modely. Každý model obsahuje informaci, o jaký typ modelu se jedná, tedy zda se jedná o model typu Basic, nebo model typu MapModel. Rovněž každý model obsahuje informaci o typu konstruktu OWL, který modeluje. Jedná-li se o třídu, jedince, nebo vlastnost. Model je jednoznačně určen svým názvem. V případě modelů typu MapModel je potřeba

udržovat i odkaz na objekt z knowlage base, pro který je model definován. A posledním členem třídy *MapModel* je samozřejmě generický list se seznamem instancí daného modelu.

Instance jsou pak reprezentovány třídou *MapModelInstance*. Každá instance je popsána svým názvem. Nese rovněž odkaz na odpovídající element z kowlage base, který reprezentuje. Součástí této třídy je seznam relací, které ukazují na vazbu mezi ostatními instancemi modelů. Tento seznam relací je tedy generický list typu *MapModelInstance*.

5.4 Řešení modelů typu Construct

Tento typ modelů se od předchozích liší. Nejedná se totiž pouze o interpretaci objektu obsaženého v ontologii a jeho grafické zobrazení. Modely typu Construct mohou být v podstatě složeny z celé řady objektů ontologie.

Proč vlastně takovýto typ modelů? Samotné vykreslení ontologie po jednotlivých prvcích často neprozradí všechny pohledy na dané objekty. Hierarchie objektů nám sice zajistí jistý vhled do uspořádání ontologie a v mnoha případech lze z tohoto pohledu vyvodit jasné důsledky. V některých případech ale můžeme v ontologii některé vztahy přehlédnout, popřípadě můžeme chtít tyto informace zobrazit ve zřejmém kontextu, který ale jednoduchá interpretace tříd, jedinců a vlastností neumožní.

Představme si pomyslnou ontologii, ve které jsou definovány třídy *Clovek* a *Manzelstvi*. Instancemi těchto vytvoříme soubor jedinců, které mezi sebou navzájem propojíme vlastnostmi *JeVManzelstvi* a *VzniklVMazelstvi*. Tyto vlastnosti nám korespondují rodinné vztahy a celá ontologie může představovat jakýsi pomyslný rodokmen. Analogicky bychom si mohli tento příklad představit například na vztazích ve firmě. Ontologie nám nabízí jedinou možnost jak ilustrovat takovéto vztahy a to pomocí změti jedinců a jejich vztahů v návaznosti na dané třídy. Předpokládejme, že jiné uspořádání ontologie není možné. V tuto chvíli nám nezbývá nic jiného než navrhnout vlastní způsob zobrazení těchto rodinných vztahů.

Pro takovéto případy byl vytvořen model Construct, který umožňuje si nad vybranou doménou, tedy třídou, definovat jisté podmínky, pomocí kterých se do modelu zahrnou jen ty instance tříd, které skutečně chceme zobrazit.

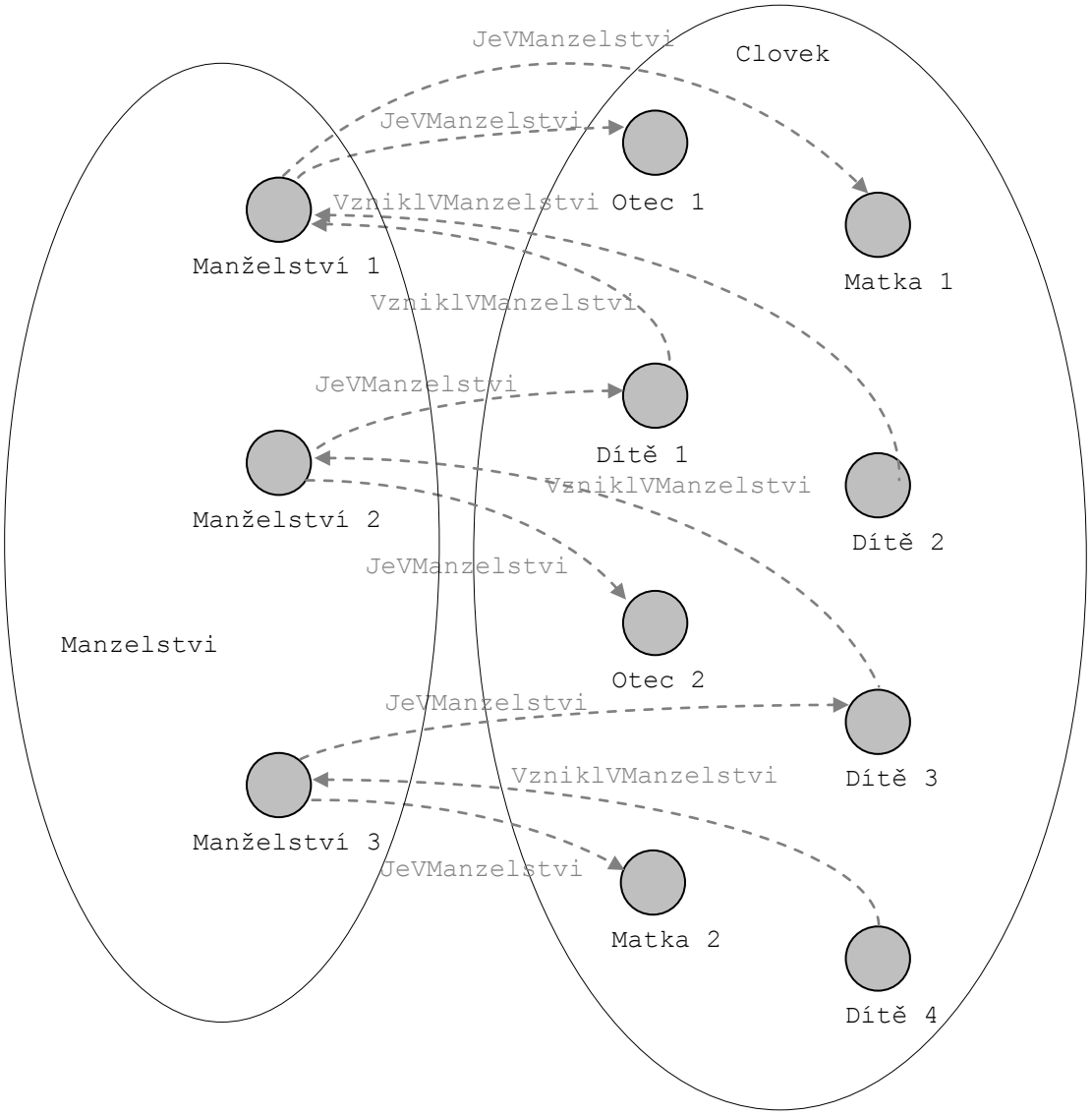
5.4.1 Stavba modelu a tvorba instancí

Vytváření modelů tedy spočívá ve volbě domény modelu z řady definovaných tříd v ontologii. Tuto doménu budeme nazývat klíčem modelu a třídu zvolenou za doménu pak klíčovou třídou. To samozřejmě náš problém nikterak neřeší. Proto, aby byl model funkční, je třeba ještě vytvořit pravidla, která nám zajistí výběr jen těch instancí z klíčové třídy, které skutečně chceme. Model jako takový je pak tvořen pouze klíčovou třídou a souborem pravidel, jež model rozšiřují. Ovšem samotný model netvoří výsledné grafické komponenty. Těmi jsou až instance daného modelu.

Tento typ modelování má nejvyšší prioritu a modely daného typu jsou tedy vyhodnocovány jako první. K opodstatnění, proč tomu tak je, si musíme představit výsledné zobrazení všech modelů. Výsledné zobrazení se totiž omezí jen na ty modely, jejichž jedinci nejsou obsaženi v instancích

modelů typu Construct jako instance klíčové třídy. Pokud by tomu tak nebylo, zobrazili by se tito jednotlivci v modelu dvakrát, či vícekrát. Což by jednak zaneslo do výsledného zobrazení zmatek a mimo jiné by se v konečném důsledku mohly stejné instance vlastností opakovat. Samotnou tvorbu modelů typu Construct si můžeme demonstrovat na následujícím motivačním příkladu založeném na výše zmíněné pomyslné ontologii rodinných vztahů.

Příklad 2: Mějme tedy ontologii, ve které jsou definovány třídy *Clovek* a *Manzelstvi*. Tyto třídy budou obsahovat celou řadu instancí tříd, které jsou zobrazeny na obrázku 8. V této ontologii jsou definovány vlastnosti *JeVmanzelstvi* a *VzniklVManzelstvi*. Doménou vlastnosti *JeVManzelstvi* je třída *Manzelstvi*, rozsahem vlastnosti je třída *Clovek*. Doménou vlastnosti *VzniklVManzelstvi* je třída *Clovek*, rozsahem vlastnosti je třída *Manzelstvi*. Jednotlivé souvislosti jedinců jsou patrné z obrázku 9.



Obrázek 9: Vztahy mezi objekty pomyslné ontologie rodinných vztahů

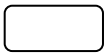
Po letmém pohledu na zobrazení této ontologie je jasné, že rodinné vztahy nejsou zcela patrné a v případě jiného rozložení objektů by zcela nešly vyčíst. Proto se pokusíme vytvořit modely typu Construct, které by jasně definovaly rodinu a rodinné vztahy. Model Rodina bude složen pouze z jedinců náležících do manželství. Model Rodinný vztah nám bude ukazovat na vztahy mezi jednotlivými rodinami.

Model Construct: Rodina

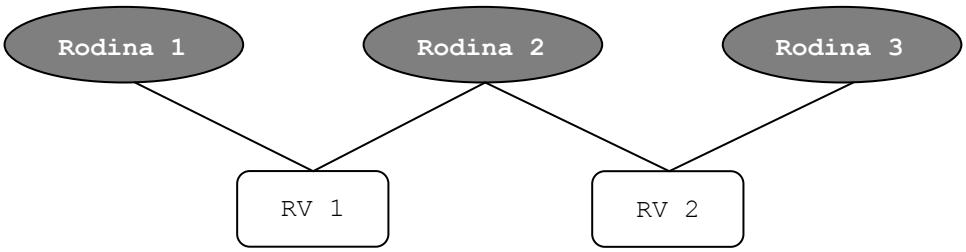
Klíčová třída: Manzelstvi
Pravidlo 1: Budou zahrnuti všichni jedinci, se kterými jsou instance klíčové třídy spojeny vlastností JeVManzelstvi
Pravidlo 2: Budou zahrnuti všichni jedinci, kteří odkazují na instancí klíčové třídy vlastností VzniklVManzelstvi
Model bude zobrazen následovně:

Model Construct: Rodinný vztah (zkraceně RV)

Klíčová třída: Clovek
Pravidlo 1: Budou zahrnuti všichni jedinci, se kterými jsou instance klíčové třídy spojeny vlastností VzniklVManzelstvi.
Pravidlo 2: Budou zahrnuti všichni jedinci, kteří odkazují na instancí klíčové třídy vlastností JeVManzelstvi.
Model bude zobrazen následovně:



Takto jsme si tedy definovali dva modely typu Construct, které jsou vymezeny danými pravidly. V následujícím obrázku 10 je zobrazeno vykreslení modelů pro porovnání s původním vykreslením ontologie.



Obrázek 10: Vykreslení modelů typu Construct

Výsledkem je tedy vykreslení modelů, ze kterých je již patrný rodokmen uložený v dané ontologii. Nicméně se nám vytratila informace o jedincích a jejich doménách obsažených v daných modelech. Proto je potřeba zobrazit tyto informace v rámci aplikace.

5.4.2 Omezení a rozšíření modelu

V předchozím textu jsme naznačili, že v rámci vytváření modelu je potřeba definovat jistá pravidla. Bez nich by totiž model typu Construct byl vykreslen stejně jako model typu MapModel, jenž by byl definován pro konkrétní třídu.

Tyto pravidla nám do jisté míry omezují výslednou množinu instancí modelu, ale ve své podstatě jej rozšíří o jedince spojené zvoleným pravidlem s daným modelem. Proto jsme pro ně zvolili jednotné označení Restriction v rámci implementace. Tyto pravidla jsou vždy definována pro objektové vlastnosti definované v ontologii a jejich vyhodnocování je závislé na pořadí, ve kterém byly definovány.

V naší implementaci jsou realizovány pouze dva typy pravidel.

- All:individuals assertions with – představuje vazbu, kde ve zvolené vlastnosti vystupuje instance klíčové třídy jako jedinec, pro nějž byla instance objektové vlastnosti definována (jedná se o object property assertions daného jedince klíčové třídy)
- All:individuals with – představuje vazbu, kde ve zvolené vlastnosti vystupuje instance klíčové třídy jako jedinec spojený s jiným jedincem přes objektovou vlastnost, která byla definována pro tohoto jedince (jedná se o object property assertions daného jedince a instance klíčové třídy je na pravé straně této vazby)

Klíčové slovo All v typu pravidla naznačuje, že budou do instance modelu přidáni všichni jedinci splňující dané pravidlo.

Vyhodnocení instancí modelu je pak tvořeno každou instancí klíčové třídy, která odpovídá daným pravidlům. To znamená, že za instanci modelu bude považována jen taková instance klíčové třídy, která splňuje všechny pravidla a pro každé pravidlo existuje alespoň jeden jedinec splňující dané pravidlo.

5.4.3 Vztahy mezi instancemi modelu

Na obrázku 10 lze vidět vytvořené vazby mezi jednotlivými instancemi modelů typu Construct. Tyto vazby se vyhodnocují na rozdíl od předchozích typů modelů jiným způsobem. Vazby jsou zde tvořeny na základě obsažených objektů v instanci modelu.

Pokud dvě instance různých modelů obsahují totožného jedince, je mezi těmito instancemi vytvořena vazba. Na obrázku 10 jsou vazby tvořeny mezi instancemi modelu Rodina a Rodinný vztah. Speciálním příznakem se ale dá vyhodnocování relací pro daný model potlačit. Pokud se tak učiní, nebudou pro žádnou instanci tohoto objektu vyhodnocovány relace, které by měli být vytvořeny směrem od tohoto modelu k jiným modelům. Podíváme-li se zpětně na definici těchto modelů a definovanou ontologii na obrázku 9, zjistíme, že společným jedincem pro instanci Rodina 1 a instanci RV 1 bude jedinec *Dítě 1*. Tento jedinec je obsažen v obou modelech. V první instanci Rodina 1 díky splnění 2. pravidla, které do modelu přidává všechny jedince spojené s instancí klíčové třídy vlastností *VzniklVManzelstvi*. V druhé instanci RV 1 je tento jedinec obsažen právě jako klíčová instance třídy *Clovek*.

5.4.4 Implementace modelů

Implementace modelů typu Construct je v našem frameworku obsažena v balíčku *Construct*. Tento balíček obsahuje všechny třídy, které vytvářejí a vyhodnocují modely tohoto typu. Základní třídou je *ConstructBase*, která především udržuje seznam všech vytvořených modelů. Tato třída rovněž vyhodnocuje vztahy mezi modely pomocí výše zmíněného postupu a to metodou *ResolveModelsRealtions*.

Další třídou obsaženou v balíčku Construct je *ConstructModel*, která reprezentuje vytvořené modely. Mimo popisného atributu pro název modelu je ve třídě uložen odkaz na instanci klíčové třídy v knowlage base. Jsou zde rovněž uloženy zadané pravidla – restriction. A samozřejmě nesmí chybět ani generický seznam typu *ConstructInstance*,ve kterém se udržují jednotlivé instance daného modelu. V této třídě je mimo jiné definována metoda *ResolveModel*, která vyhodnocuje daný model a vytváří instance modelu.

Třída *ConstructInstance* pak představuje jednu konkrétní instanci modelu. Obsahuje instanci klíčové třídy jako odkaz na jedince z knowlage base a generické seznamy objektů z knowlage base, které daná instance obsahuje. Je zde i generický seznam relací datového typu *ConstructInstance*. Ten představuje odkazy na jiné instance modelů, se kterými má daná instance vytvořený vztah.

Poslední podstatnou třídou v tomto balíčku je *ConstructRestriction*, která představuje instance pravidel definovaných v modelu. Obsahuje textový výraz, který reprezentuje zápis daného pravidla a odkaz na objektovou vlastnost z knowlage base, se kterou je restricton spojena. Po vytvoření objektu restriction je automaticky validován textový výraz expresion a v případě správného zadání jsou algoritmicky doplněny popisné informace o daném pravidlu.

5.5 Vyhodnocování vztahů modelů různých typů

Vztahy mezi modely různých typů jsou vyhodnocovány až ve třídě *OWLApiModels*, bezprostředně před tím, než mají být modely vykresleny. Vztahy mezi modely typu Basic a MapModel jsou již vyhodnoceny na úrovni třídy *MapModelBase*, jak bylo popsáno výše. Nicméně vztahy mezi modely Construct a ostatními typy modelů mají svá vlastní pravidla vyhodnocování.

Modely typu Construct obsahují ve svých instancích více objektů z dané ontologie. Vztahy se proto řeší pouze pro instanci klíčové třídy a to tak, že za vztah mezi instancí modelu typu Construct jsou považovány vlastnosti, ve kterých tato instance vystupuje. Jedná se tedy skoro o analogický přístup vyhodnocování vazeb, který platí pro modely typu Basic a MapModel.

Například pokud je klíčová instance modelu Construct spojena vlastností s jiným jedincem, který je definován modelem Basic a daná vlastnost má rovněž vytvořen svůj model, pak je mezi těmito modely a modelem vlastnosti vytvořena spojitost.

6 Grafické rozhraní aplikace

Vytvoření frameworku zpracovávajícího ontologie a vytvářejícího modely by jistě nestačilo pro ověření funkčnosti frameworku a úspěšné dokončení této práce. Proto bylo současně implementováno i grafické uživatelské rozhraní aplikace, které aplikuje vyvinutý framework. Na obrázku 6 si můžeme všimnout, že grafický editor ontologií se skládá ze dvou částí – frameworku a uživatelského rozhraní. V tomto uživatelském rozhraní jsou interpretovány data z načtené ontologie, tvoří se v něm výběry objektů pro modelování, vytváří se modely, přiřazují se těmto modelům grafické interpretace a samozřejmě se i vykreslují výsledné modely jako grafické komponenty.

6.1 Technologie

Na úvod je třeba jen ve stručnosti zmínit využité technologie pro tvorbu uživatelského rozhraní. Pro návrh GUI byla vybrána knihovna uživatelských prvků **Swing**, která je určena pro platformu Java. Z této knihovny je využita celá řada prvků. Těmi jsou především kontejnery JFrame, které tvoří okna uživatelského rozhraní. Téměř všechny okna aplikace jsou tvořeny tímto prvkem. Z dalších využitých prvků zmíníme JTree, který slouží k zobrazení hierarchické stromové struktury. Ten je využit především na zobrazení prvků z knowlage base jako je například hierarchický seznam tříd.

S knihovnou Swing jsme si bohužel nevystačili. Pro zobrazení zpracovaných prvků ontologie a modelování komponent byl sice Swing dostačující, problém ale nastal při hledání vhodného nástroje pro vykreslování grafických komponent definovaných modely. Po delším hledání se jako nejvhodnější nástroj jevila knihovna Draw2d určená k vykreslování grafů a grafických komponent. Ta je ale postavena na knihovně SWT, která je rovněž knihovnou grafických prvků určených pro uživatelské rozhraní. Proto bylo nutné do naší aplikace přidat nový formulář navržený v knihovně SWT.

SWT je tedy alternativou ke knihovně Swing. K vykreslování prvků uživatelského rozhraní využívá, na rozdíl od Swingu, nativních knihoven operačního systému. Z knihovny SWT byly využity především prvek Shell, který v SWT představuje okno aplikace.

Zmíněný nástroj **Draw2d** je jednoduchá knihovna grafických komponent, které jsou zobrazeny na plátně z knihovny SWT. Lépe řečeno na plátně, které dědí své vlastnosti z SWT plátna. Všechny grafické prvky jsou nezávislé na operačním systému, na rozdíl od prvků knihovny SWT. Z této knihovny jsme následně využili některé grafické komponenty a především objekty, které vytváří spojnice mezi těmito grafickými komponentami.

6.2 Popis uživatelského rozhraní

Nyní se již v popisu práce dostáváme k samotnému uživatelskému rozhraní. Uživatelské rozhraní je rozděleno do tří částí, nebo možná lépe řečeno, do tří režimů zobrazení. Každý režim zobrazení má svá specifika a každý z těchto režimů zpřístupňuje jinou část práce s grafickým editorem ontologií.

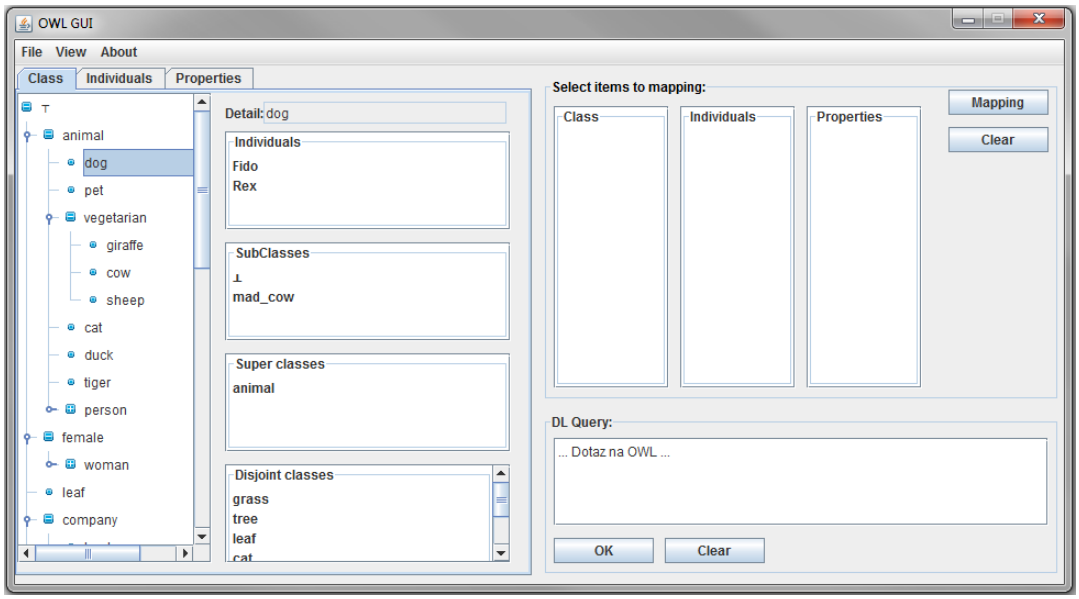
Těmito třemi režimy jsou:

- Ontologický pohled
- Modelovací pohled
- Grafický pohled

6.2.1 Ontologický pohled

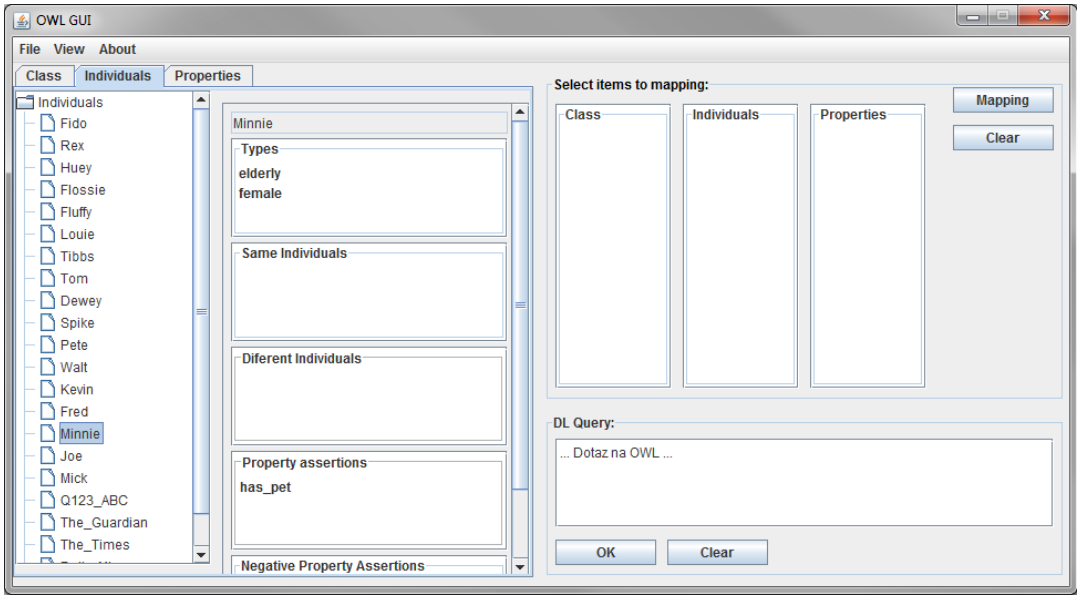
Ontologický pohled nám nabízí náhled na zpracovanou ontologii. V tomto režimu aplikace se ontologie načítá a zpracovává. Rovněž se v tomto režimu zobrazují prvky obsažené v dané ontologii. K zobrazení ontologických prvků se využívá načítání objektů z knowlage base našeho frameworku.

Ontologické třídy jsou zobrazeny ve stromové struktuře, tak jak jsou definovány v samotné ontologii. Pro každou vybranou třídu jsou zobrazeni jedinci jako její instance, její podtřídy, její nadtřídy a všechny disjunktní třídy. Na následujícím obrázku 11 lze na kartě Class vidět zobrazení tříd z ontologie People.



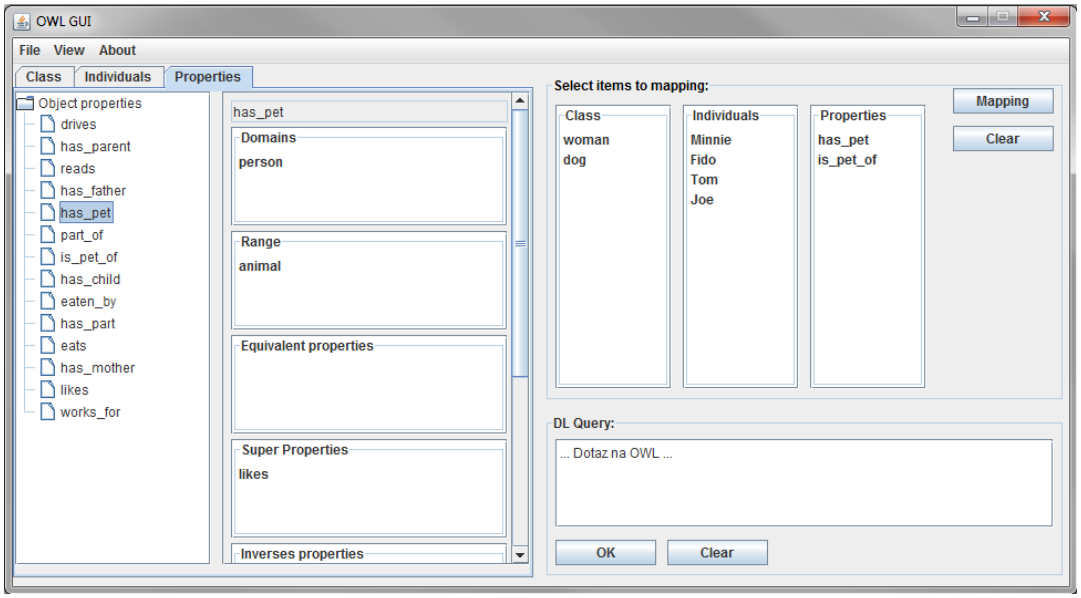
Obrázek 11: GUI - Ontologický pohled - přehled tříd

V rámci ontologického pohledu je na samostatném přehledu dostupný i seznam jedinců obsažených v ontologii. Každý jedinec zde má dostupné informace, které jdou z ontologie vyčíst. Pro vybraného jedince můžeme zjistit, jaký má definovaný typ ontologické třídy, jejíž může být instancí. Vidíme, zda je tento jedinec definován jako totožný, nebo rozdílný s jiným jedincem. A pochopitelně je zde i zobrazeno, jaké vlastnosti a negativní vlastnosti jsou pro tohoto jedince vytvořeny. Na obrázku 12 vidíme na záložce Individuals seznam jedinců z ontologie People a přehled informací o vybraném jedinci.



Obrázek 12: GUI - Ontologický pohled - přehled jedinců

Přehled objektových vlastností je vytvořen analogicky k předchozím dvěma přehledům v ontologickém pohledu. Každá vlastnost definovaná v ontologii je zobrazena v tomto přehledu. Mimo seznamu vlastností jsou zde zobrazeny i informace dostupné pro vybranou vlastnost. Můžeme zde zjistit, jaké doménové třídy jsou definovány pro danou vlastnost, popřípadě které třídy tvoří rozsah dané vlastnosti. Jsou zde zobrazeny i ekvivalentní, inverzní a disjunktní vlastnosti. Nechybí ani informace o hierarchii vlastností a tedy zde nalezneme i nadřazené a podřazené vlastnosti. Na obrázku 13 je zobrazen seznam objektových vlastností ontologie People.



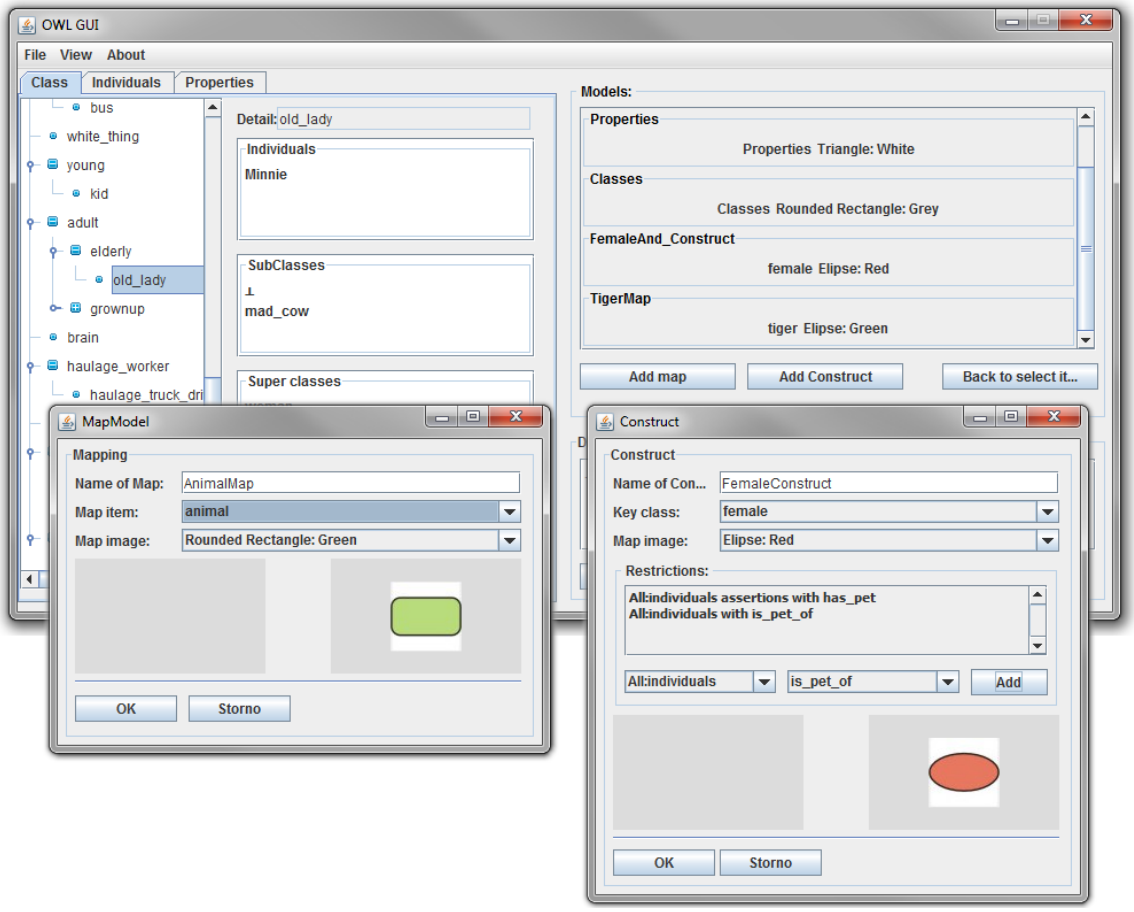
Obrázek 13: GUI - Ontologický pohled - přehled vlastností

Ontologický pohled ale neslouží jen pro procházení a zobrazování ontologických prvků. Zároveň se již v tomto pohledu vytváří výběry ontologických objektů pro vytváření modelů. V tomto režimu zobrazení je aplikace rozdělena na dvě logické části. První část, jak již bylo výše popsáno, slouží k náhledu na ontologii. Druhá část slouží právě k zobrazení vybraných prvků pro tvorbu modelů. Každý zobrazený prvek z ontologie je přidán pro výběr dvojklikem na jeho název v detailu zobrazení tohoto prvku. Na obrázku 13 je tento výběr patrný v pravé části aplikace.

6.2.2 Modelovací pohled

Modelovací pohled, jak už název napovídá, slouží k vytváření modelů grafických komponent z vybraných prvků ontologie. Jak již bylo popsáno v předchozí kapitole věnované právě vytváření modelů, slouží tyto modely především k vytvoření vlastních obrazů objektů v ontologii.

Z vybraných ontologických prvků pro mapování se vytváří modely trojího typu. Modely Basic a MapModel jsou v aplikaci vytvářeny pomocí dialogového okna MapModel. toto dialogové okno je dostupné v modelovacím pohledu po kliknutí na tlačítko Add map. Modely Construct se vytváří v dialogovém okně Construct dostupném po kliknutí na tlačítko Add Construct. Na obrázku 14 jsou ukázány oba tyto dialogy i vzhled aplikace v modelovacím pohledu.

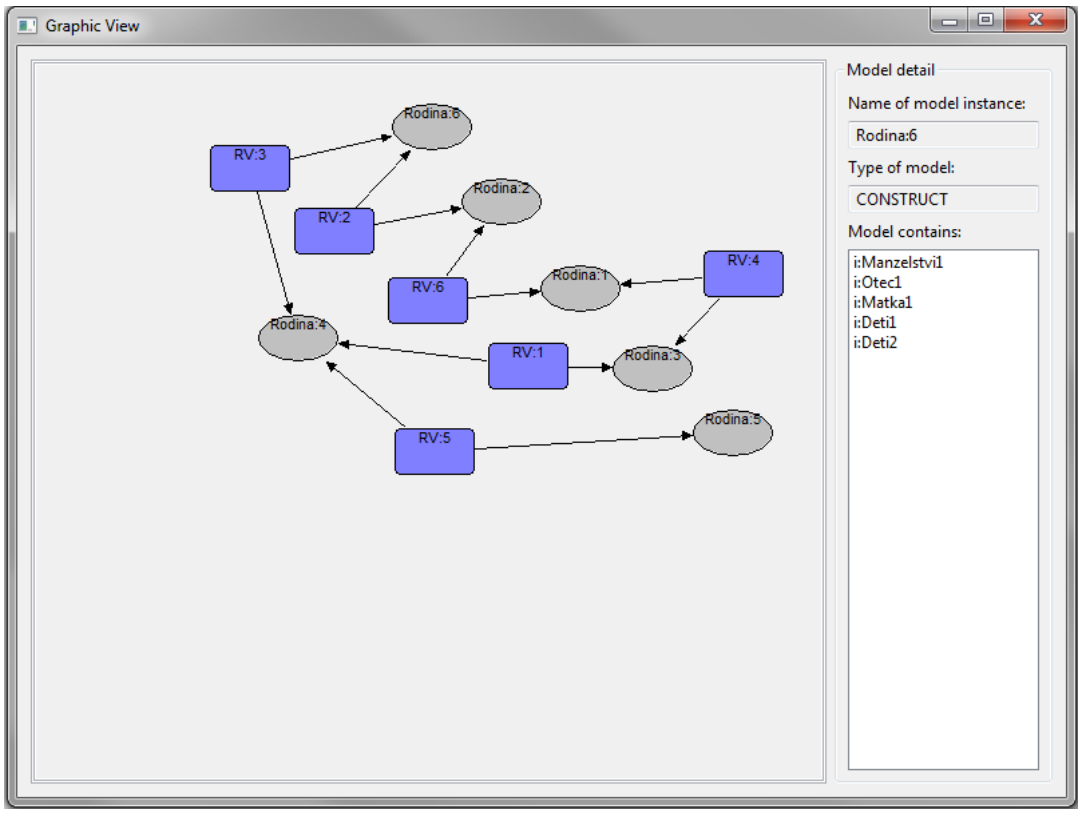


Obrázek 14: Ukázka modelování z aplikace

6.2.3 Grafický pohled

Pro vytvoření modelů je jejich zobrazení a vykreslení do aplikace dostupné v grafickém pohledu. Grafický pohled se zobrazí po vybrání položky menu View – Graphic View v aplikaci. Namodelované grafické komponenty jsou v tu chvíli vyhodnoceny a vykresleny na plátno. Mezi grafickými komponentami jsou vykresleny i jejich vztahy pomocí směrových spojnic. Spojnice jdou vždy směrem od modelu, který tvoří relaci, k modelu, se kterým je touto relací spojen. S grafickými komponentami jde samozřejmě hýbat přetahováním, tedy metodou drag and drop.

V pravé části tohoto zobrazení se nacházejí informace o instancích grafické komponenty. Po kliknutí na vybranou komponentu se tyto informace automaticky překreslí. Nalezneme zde název zvolené instance modelu, typ modelu a soupis ontologických objektů, které se v dané instanci nacházejí. Na obrázku 15 je uveden jako příklad vykreslení ontologie rodinných vztahů. Ontologie je podobná té, o které jsme hovořili v kapitole 5.4.1.



Obrázek 15: GUI - Grafický pohled - ukázka vykreslení

6.3 Grafická reprezentace modelů

Jak bylo vidět na ukázkách vytváření mapování na obrázku 14 i na ukázce grafického pohledu na obrázku 15, kde jsou vykresleny grafické komponenty, modelům jsou zadávány grafické reprezentace

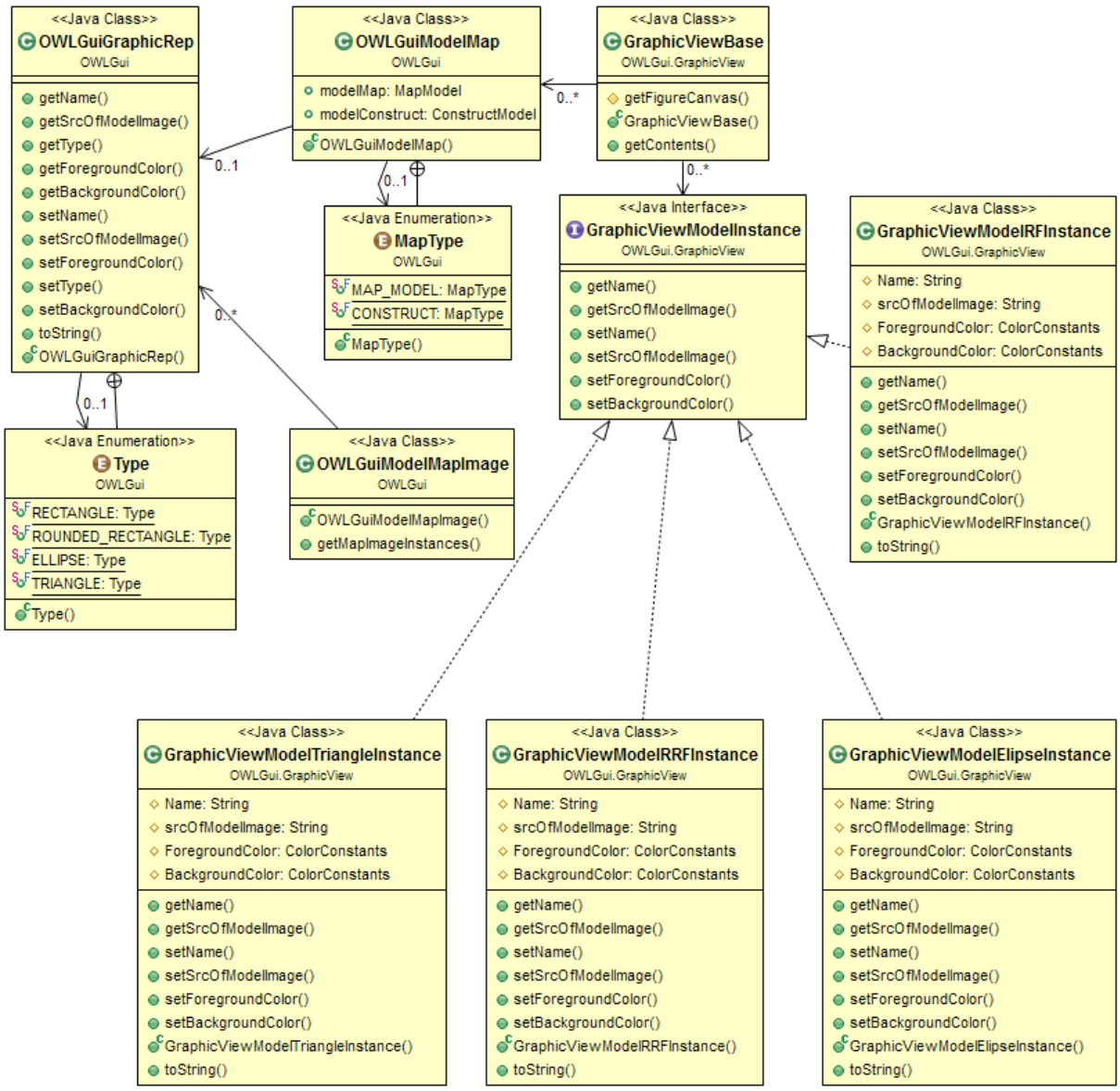
definovaných tvarů a barev. Tím se vytváří grafické komponenty v naší aplikaci. Na rozdíl od samotných modelů se tyto grafické reprezentace nevytvářejí a neuchovávají ve frameworku. To je pochopitelné, framework se má starat pouze o interpretaci dat z ontologie a vytváření modelů. Samotná interpretace modelu v grafickém rozhraní je pak na samotné aplikaci, která modely vykresluje.

V naší aplikaci, jak bylo zmíněno výše, je využita knihovna Draw2d pro vykreslování. Tato knihovna je vhodná pro účely vytváření grafických komponent především proto, že její grafické prvky jsou již vytvořeny jako prvky grafů. To znamená, že podporují vytváření spojení mezi sebou. Pro spojení jsou v knihovně vytvořeny speciální objekty, kterým stačí deklarovat, mezi kterými prvky má být spojnice vytvořena.

Základní třídou pro práci s grafickými komponentami je třída *OWLGuiModelMap*, ve které se ukládají vytvořené grafické komponenty spolu s odkazem na jejich grafickou interpretaci. Tuto interpretaci v aplikaci tvoří třída *OWLGuiGraphicRep*, která uchovává informace o jednotlivých reprezentacích. Udržuje informaci o tvaru objektu, barvě popředí, barvě pozadí, barvě okraje a také odkaz na reprezentativní obrázek, který je využit v aplikaci při modelování. V aplikaci samotné je pak vytvořená statická třída *OWLGuiMapImage*, která definuje dostupné grafické interpretace právě z předešle jmenované třídy. Třída *OWLGuiModelMap* ale nedefinuje jak vykreslit grafickou komponentu na plátno.

Pro potřeby vykreslování grafických komponent jsme vytvořili nový balíček *GraphicView*. Základem tohoto balíčku je třída *GraphicViewBase*, která udržuje mapu vytvořených instancí třídy *OWLGuiModelMap*, tedy instance definovaných grafických komponent. Samotná třída pak zajišťuje vykreslení těchto komponent metodou *getContents*. Každý tvar grafické komponenty má v balíčku GraphicView zastoupení svou adekvátní třídou. Interface *GraphicViewModelInstance* je společným rozhraním pro jednotlivé třídy vykreslující dané tvary. Díky tomu je možné v třídě *GraphicViewBase*, držet mapu vykreslených grafických komponent.

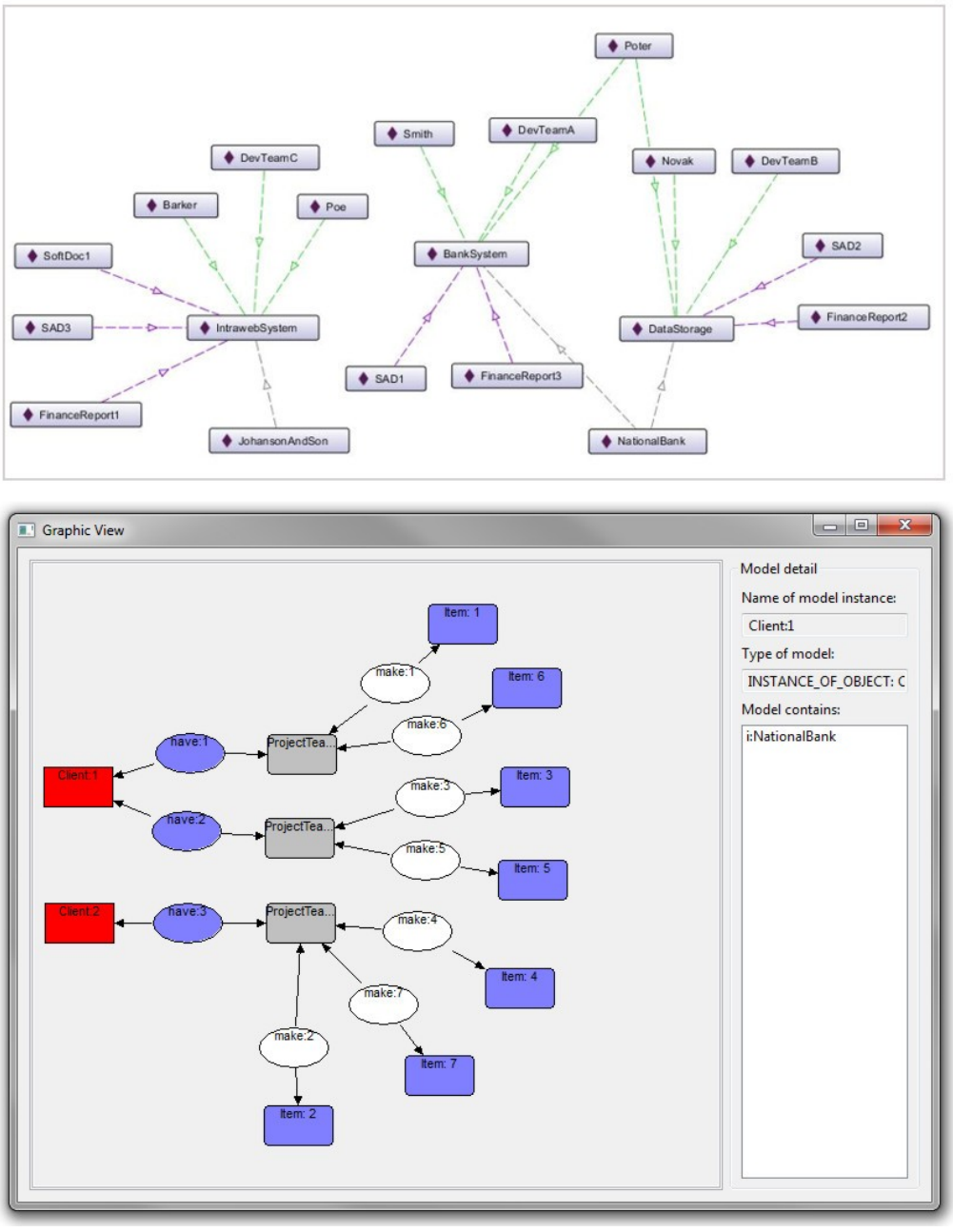
Pro správné porozumění vztahů mezi jmenovanými třídami si nyní uvedeme třídní diagram, který předchozí text doplní. Diagram obsahuje všechny třídy využité k vytváření a vykreslování grafických reprezentací modelů. Tento třídní diagram byl na rozdíl od diagramu na obrázku 6 vygenerován pomocí automatického generátoru třídních diagramů ObjectAid UML Explorer (13) ve vývojovém prostředí Eclipse.



Obrázek 16: UML třídní diagram tříd pro vytváření grafických komponent

7 Porovnání grafických výstupů

V této závěrečné kapitole se pokusíme srovnat grafickou interpretaci ontologií v naší aplikaci s grafickým zobrazením v editoru Protégé. Na následujícím porovnání na obrázku17 je vykreslena ontologie CompanyStructure, která byla vytvořena za účelem testování. Tato ontologie simuluje jednoduchou strukturu organizace, která se zabývá vývojem softwaru. Ontologie obsahuje objekty popisující organizační strukturu, projekty, dokumenty a zákazníky.



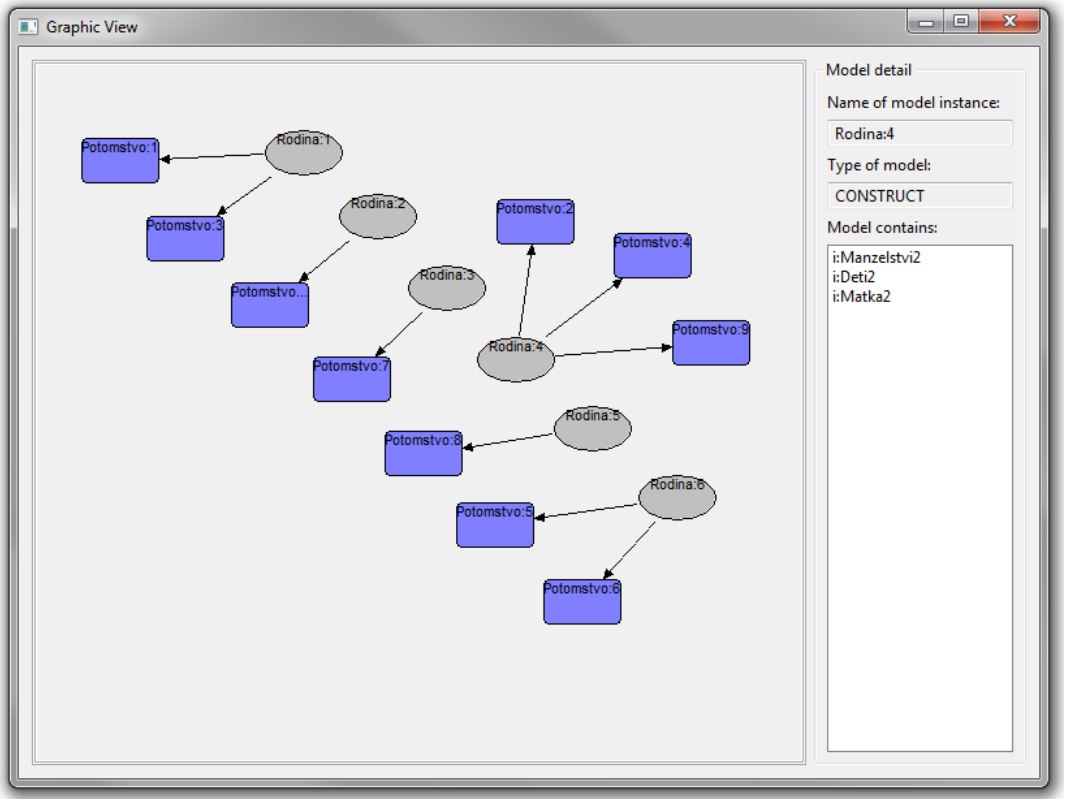
Obrázek 17: Porovnání grafických zobrazení

Na obou grafech je tedy vykreslena stejná ontologie. První, v horní části obrázku 17, je vizualizace, kterou vykreslí editor Protégé v části OntoGraf. Pro danou ukázku jsme vynechali vykreslení tříd a jednotlivé objekty se pokusili co nejpřehledněji seřadit. V druhé, dolní části obrázku 17, je zobrazena ontologie jako vykreslení grafických modelů, které byly sestaveny následujícím způsobem:

- Model ProjectTeam typu Construct – modeluje jednotlivé projekty a všechny jedince z různých organizačních struktur, kteří na něm pracují. K tomu je využita vlastnost *WorkFor*.
- Model Client typu MapModel – modeluje třídu *Client*.
- Model Item – modeluje všechny nedefinované jedince z výběru prvků pro modelování.
- Model have – modeluje vlastnost *IsOwner* mezi jednici typu *Client* a *Project*.
- Model make – modeluje vlatnost *CreateFor* mezi jedinci typu *Items* a *Project*.

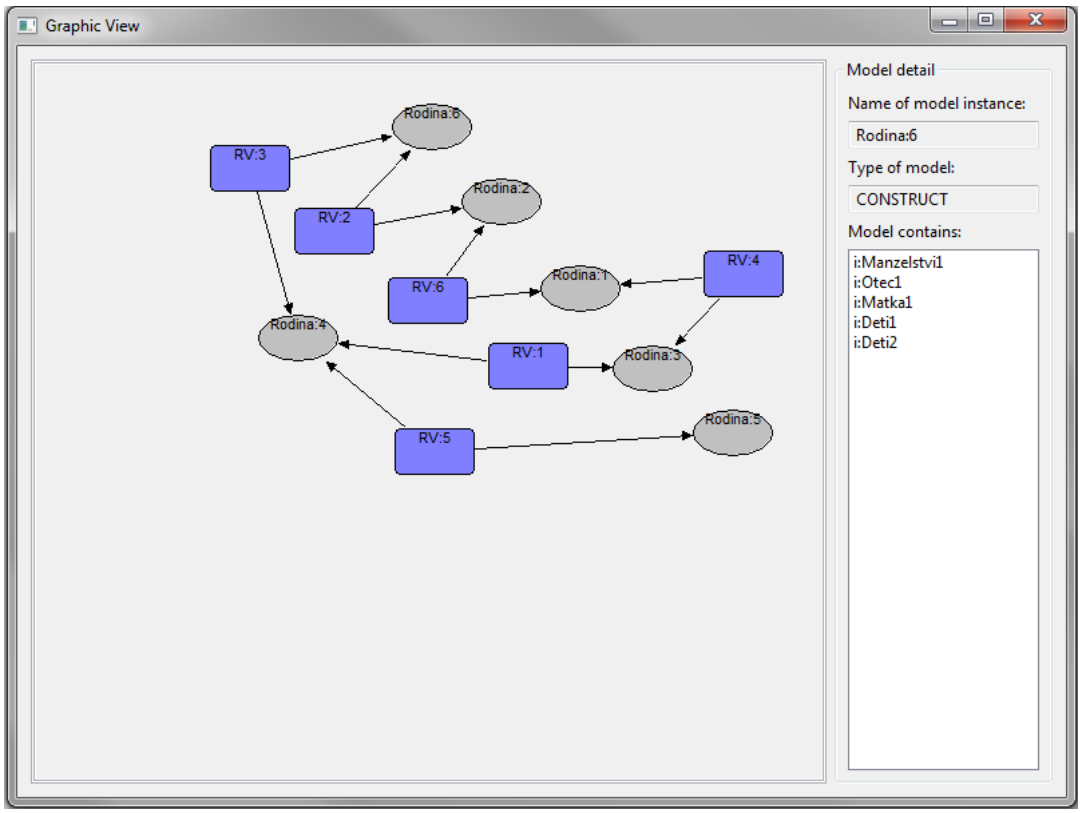
Namodelované zobrazení tedy zjednodušuje ontologii pouze na projekty, jejich vlastníky a předměty, které s projektem souvisí. Navíc jsou vykreslené objekty ontologie barevně a tvarově rozlišeny. Ve srovnání s vykreslením z editoru Protégé se tedy jedna o zcela alternativní a svým způsobem i přehlednější zobrazení stejné ontologie.

Na následujících obrázcích 18 a 19 je zobrazena totožná ontologie, v každém zobrazení byl ale využit jiný přístup k vytváření modelu typu Construct. Tímto je demonstrováno možné využití jedné a té samé ontologie při dvou různých pohledech na ni. K vykreslení byla opět použita zmíněná testovací ontologie Rodinných vztahů z kapitoly 5.4.1.



Obrázek 18: Ontologie rodinných vztahů vykreslující Rodinu a její potomstvo

Na obrázku 18 je ontologie vykreslena pomocí dvou modelů typu Construct. První model zobrazuje pouze rodinu, jako instance třídy *Manzelstvi* spolu s jedinci třídy *Clovek*, kteří jsou s těmito instancemi svázaní vlastností *JeVManzelstvi*. Druhý model zobrazuje potomstvo daných rodin. Tento model byl vytvořen pro všechny instance třídy *Clovek*, které byly omezeny pouze na ty, které jsou vlastností *VzniklVManzelstvi* svázány s třídou *Manzelstvi*. U tohoto modelu bylo navíc potlačeno vykreslení relací.



Obrázek 19: Ontologie rodinných vztahů vykreslující Rodiny a jejich vztahy

Na obrázku 19 je ontologie rovněž vykreslena pomocí dvou modelů typu Construct. První model zobrazuje pouze rodinu, jako instance třídy *Manzelstvi* spolu s jedinci třídy *Clovek*, kteří jsou s těmito instancemi svázaní vlastnostmi *JeVManzelstvi* a *VzniklVManzelstvi*. Model tedy zobrazuje kompletní rodinu. Druhý model zobrazuje vztahy, které mezi těmito rodinami jsou. Model vykresluje instance třídy *Clovek*, které mají omezení na vlastnosti *VzniklVManzelstvi* a *JeVManzelstvi* související s třídou *Manzelstvi*.

8 Závěr

Tato práce se zabývá teoretickým použitím ontologií k jejich alternativní vizualizaci ve vlastním grafickém editoru ontologií. Za tímto účelem bylo potřeba navrhnout a implementovat vlastní framework, který by ontologie dokázal zpracovat a načíst z nich potřebné informace a tyto informace uměl dále distribuovat. Zároveň s frameworkem vznikl i nový přístup k interpretaci informací obsažených v ontologii. Součástí práce bylo i navrhnou a implementovat přístup, který by umožňoval modelovat ontologické objekty a pomocí nich vytvářet nové objekty, tzv. modely. Ve frameworku vytvořeným modelům je pak přiřazená vizuální podoba. Spolu s ní modely vytváří grafické komponenty vhodné k vykreslení. Výsledkem této práce je implementovaný framework, který je rozšířen o aplikační část umožňující jej plně využívat pro načtení a zobrazení ontologických objektů, vytváření modelů a zobrazování vytvořených modelů. Práce na tomto zadání byla rozdělena mezi skupinu studentů, editace ontologie z grafického prostředí a její ukládání je již náplní jiné diplomové práce.

Obsahem této práce bylo především navrhnout řešení alternativní vizualizace ontologických objektů, které je řešeno tvorbou popsaných modelů a přiřazením grafické interpretace těmto modelům. V dalších krocích by bylo jistě vhodné rozšířit editor o možnost uživatelské definice vizualizací. Toho by šlo docílit zahrnutím editoru grafických komponent do implementované aplikace. Rovněž jako následující krok při vytváření modelů by mělo být prozkoumání možností deskripční logiky a zvážení jejího zapojení do vytváření modelů nad ontologiemi.

Literatura

1. **STÖRIG, Hans Joachim.** *Malé dějiny filosofie.* Berlín : Karmelitánské nakladatelství Kostelní Vydří, 2007. ISBN 978-80-7195-206-0.

2. **TANIAR, David.** *Web Semantic and Ontology.* Hershey : Idea Group Publishing, 2006. ISBN 1-59140-905-5.

3. **SVÁTEK, Vojtěch.** Ontologie a WWW. *VŠE DATAKON.* [Online] 2002. [Citace: 5. 8 2012.] http://nb.vse.cz/~svatek/onto-www.pdf.

4. **GRUBER, Thomas R.** A translation approach to portable ontology specifications. *Knowledge Acquisition.* Special issue: Current issues in knowledge modeling, 1993, Sv. 5, 2.

5. **ŠTVERÁK, Ondřej.** Ontologie pro dynamickou geovizualizaci v krizovém řízení. *Diplomová práce.* Brno : MUNI, 2009.

6. *The Syntax of CycL.* [Online] Cycorp, 28. 3 2002. [Citace: 18. 7 2012.] http://www.cyc.com/cycdoc/ref/cycl-syntax.html.

7. **ALLEMANG, Dean a HENDLER, James.** *Semantic Web for the Working Ontologist: Effective Modeling in RDFS and OWL.* Burlington : Morgan Kaufmann Publishers, 2008. ISBN 978-0-12-373556-0.

8. **FAYNBERG, Igor.** W3C. *DAML+OIL Web Ontology Language.* [Online] 12. 9 2001. [Citace: 25. 7 2012.] http://www.w3.org/Submission/2001/12/.

9. **McGUINNESS, Deborah L. a HARMELEN, Frank van.** OWL Web Ontology Language Overview. *W3C.* [Online] 12. 11 2009. [Citace: 1. 8 2012.] http://www.w3.org/TR/owl-features/.

10. **Research, Stanford Center for Biomedical Informatics.** Protégé. *What is Protégé?* [Online] 2012. [Citace: 3. 8 2012.] http://protege.stanford.edu/overview/.

11. **HORRIDGE, Matthew a BECHHOFER, Sean.** The OWL API: A Java API for OWL Ontologies. *Semantic Web Journal.* Special Issue on Semantic Web Tools and Systems, 2011, Sv. 2, 1.

12. **University of Manchester, Group.** Reasoners. *The OWL API.* [Online] [Citace: 29. 7 2012.] http://owlapi.sourceforge.net/reasoners.html.

13. **ObjectAid LLC.** The ObjectAid UML Explorer for Eclipse. *ObjectAid* . [Online] [Citace: 6. 8 2012.] http://www.objectaid.com/home.

Adresářová struktura přiloženého DVD

/img	Obrázky doplňující práci ve vysokém rozlišení
/ont	Ontologie využité k testování
/src	Zdrojové kódy diplomové práce, aplikace
/text	soubory s textem práce, zadání